

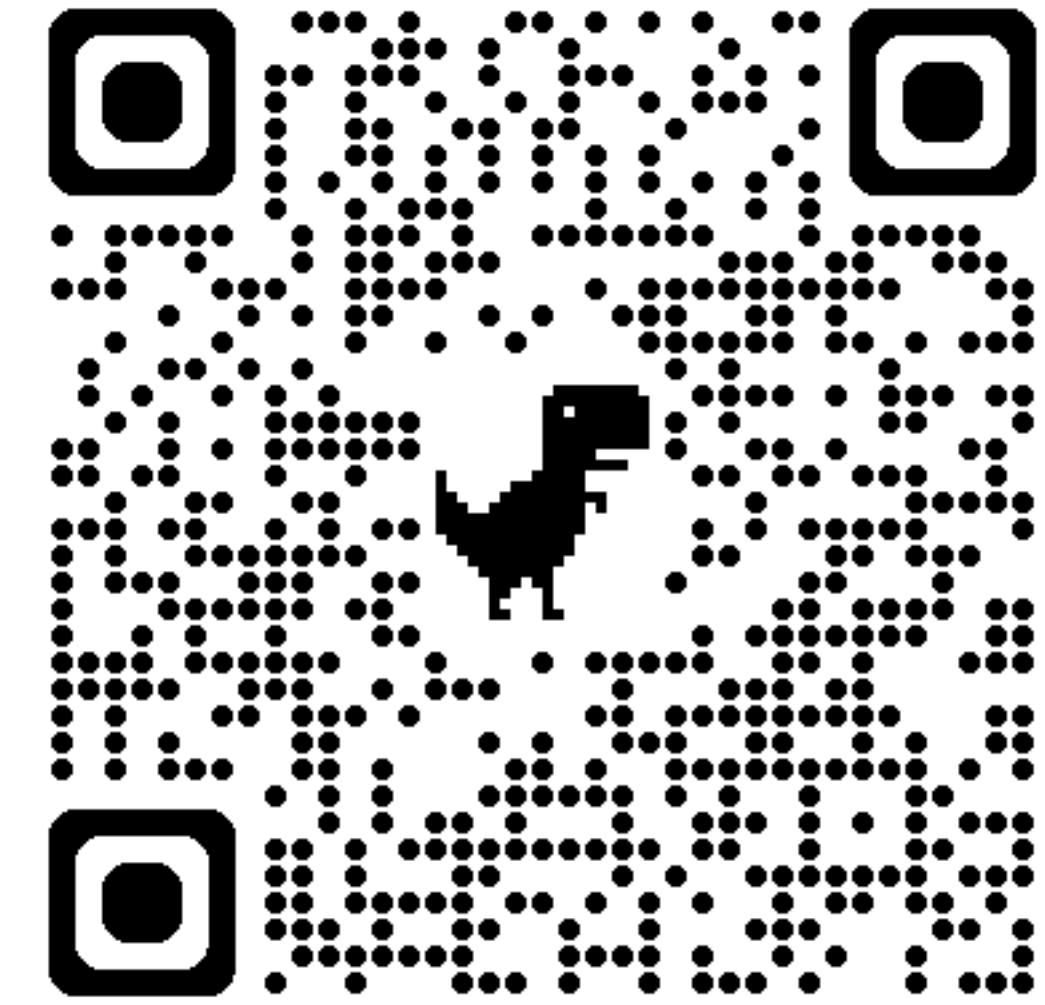
From DDD to Delivery

Daniel Hinojosa

Understanding why DDD is the cornerstone of everything

In this Presentation

- What is Domain Driven Design
- Subdomains
- Ubiquitous Language
- Bounded Context
- Integrating Bounded Contexts
- Simple Applications
- Complex Applications
- Event Storming



Slides and Material: <https://github.com/dhinojosa/from-ddd-to-delivery>

What is DDD?



What is Domain Driven Design?

- Domain Driven Design (DDD) is a set of tools that assist you in designing and implementing software that delivers high value, both strategically and tactically
- DDD focus your attention on what your company or your team should be focused on
- DDD has a strategic value in about mapping business domain concepts into software artifacts.
- DDD is about organizing code artifacts in alignment with business problems, using the same common, ubiquitous language
- DDD is about taking this business language, the ubiquitous language and creating models that represent the language and used to solve problems

Advantages & Disadvantages

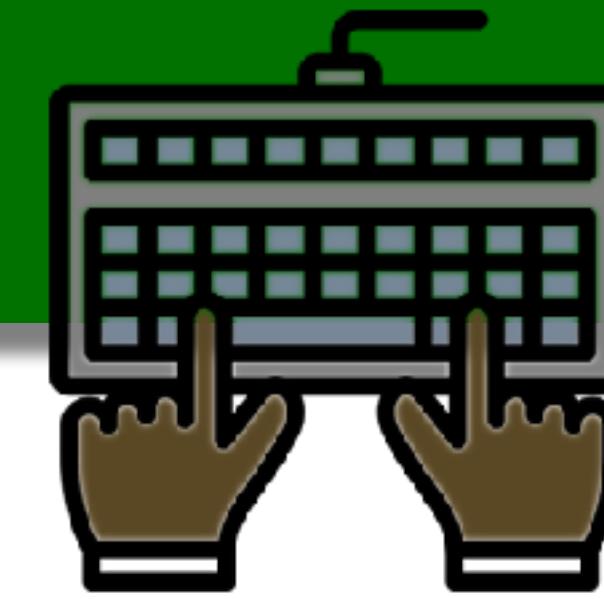
- Incremental & Flexible
- Organized: In code and teams
- Focuses on the Domain
- Learning Curve
- Time and Effort

Two Parts of DDD

Strategic



Tactical



When do I not use DDD?

- Very simple projects where the domain is one to maybe ten different models. It might be overkill to use DDD
- Regardless, thinking about domains and models is essential even if you and your team are not going through the entire DDD process
- If you are in a large company, with many business roles, then DDD will give benefit and clarity
- In the end, it depends, but you will find many aspects of DDD helpful in the large

<https://github.com/ddd-crew>

The screenshot shows the GitHub repository page for 'ddd-crew'. The top navigation bar includes a menu icon, a GitHub logo, the repository name 'ddd-crew', and various search and filter icons. Below the header, there are tabs for 'Overview' (selected), 'Repositories' (15), 'Projects', 'Packages', and 'People' (4). A profile picture of a man with glasses is visible on the right.

Domain-Driven Design Crew

Follow

4k followers Worldwide

Pinned

- welcome-to-ddd** Public
Definitions of DDD and fundamental concepts to reduce the learning curve and confusion
1.4k stars, 80 forks
- ddd-starter-modelling-process** Public
If you're new to DDD and not sure where to start, this process will guide you step-by-step
5.2k stars, 469 forks

People

Four user profile pictures are shown: a man with a beard, a man with glasses, a woman with glasses, and a man with short hair.

Top languages

HTML

Repositories

Subdomains [Strategic]



Business Domain

- Defines a company's main area of activity, it's primary business.
- Generally speaking, it's the service the company provides to its clients.
- Examples include:
 - FedEx provides courier delivery.
 - Starbucks is best known for its coffee.
 - Walmart is one of the most widely recognized retail establishments

Ancillary Domains

- A company can operate in multiple business domains.
- For example:
 - Amazon provides both retail and cloud computing services.
 - Uber is a rideshare company that also provides food delivery and bicycle-sharing services.
- It's important to note that companies may change their business domains often!

Subdomains

- For a business domain's goals and targets, a company has to operate in multiple subdomains
- A subdomain is a fine-grained area of business activity
- All of a company's subdomains form its business domain: the service it provides to its customers
- The subdomains have to interact with each other to achieve the company's goals in its business domain

Real World Subdomains

- Starbucks may be most recognized for its coffee
- The successful coffeehouse chain requires more than just knowing how to make great coffee.
- It also includes:
 - How to buy or rent real estate at effective locations
 - Hire personnel
 - Manage finances
 - Other activities

Three Types of Subdomains

Core

Generic

Supporting

Core Subdomain

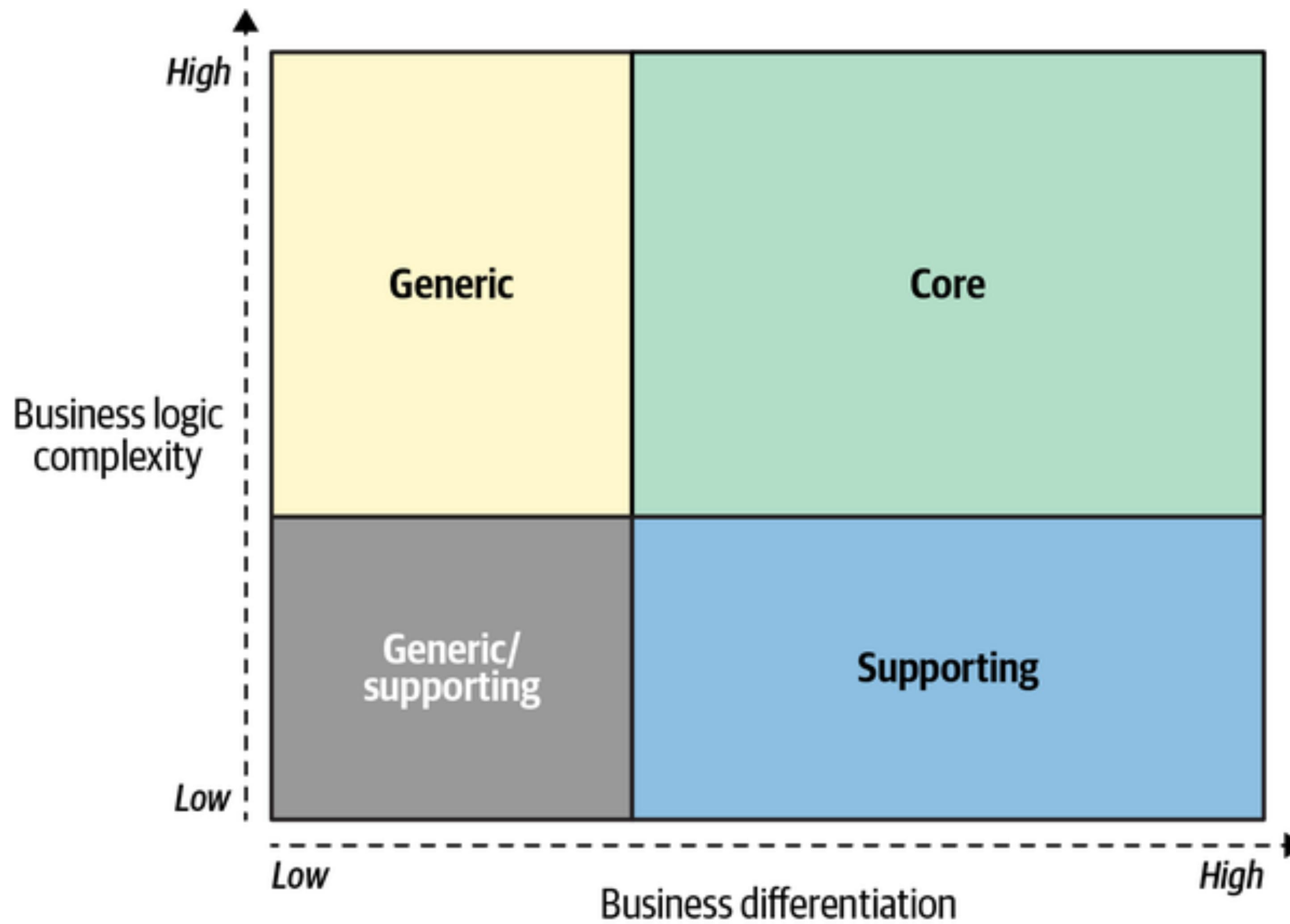
- A core subdomain is what a company does differently from its competitors
 - "Inventing new products or services"
 - "Reducing costs by optimizing existing processes"
 - "Ship products to hard-to-reach locations"
- Must be complex and hard for competitors to mimic
- It doesn't have to be technical

Generic Subdomain

- Generic subdomains are business activities that all companies are performing in the same way.
- Like core subdomains, generic subdomains are generally complex and hard to implement.
- However, generic subdomains do not provide any competitive edge for the company.
- There is no need for innovation or optimization here: battle-tested implementations are widely available, and all companies use them.
- e.g. Authenticating Users, Securing Networks, Organizing your Day, etc.

Supporting Subdomain

- Supporting subdomains support the company's business.
- However, contrary to core subdomains, supporting subdomains do not provide any competitive advantage.
- Business logic resembles ETL (extract, transform, load) operations; that is, the so-called CRUD (create, read, update, and delete) interfaces.
- These activity areas do not provide any competitive advantage for the company, and therefore do not require high entry barriers.



Subdomain type	Competitive advantage	Complexity	Volatility	Implementation	Problem
Core	Yes	High	High	In-house	Interesting
Generic	No	High	Low	Buy/adopt	Solved
Supporting	No	Low	Low	In-house/outsource	Obvious

Identifying Domain Experts [Strategic]



Domain Experts

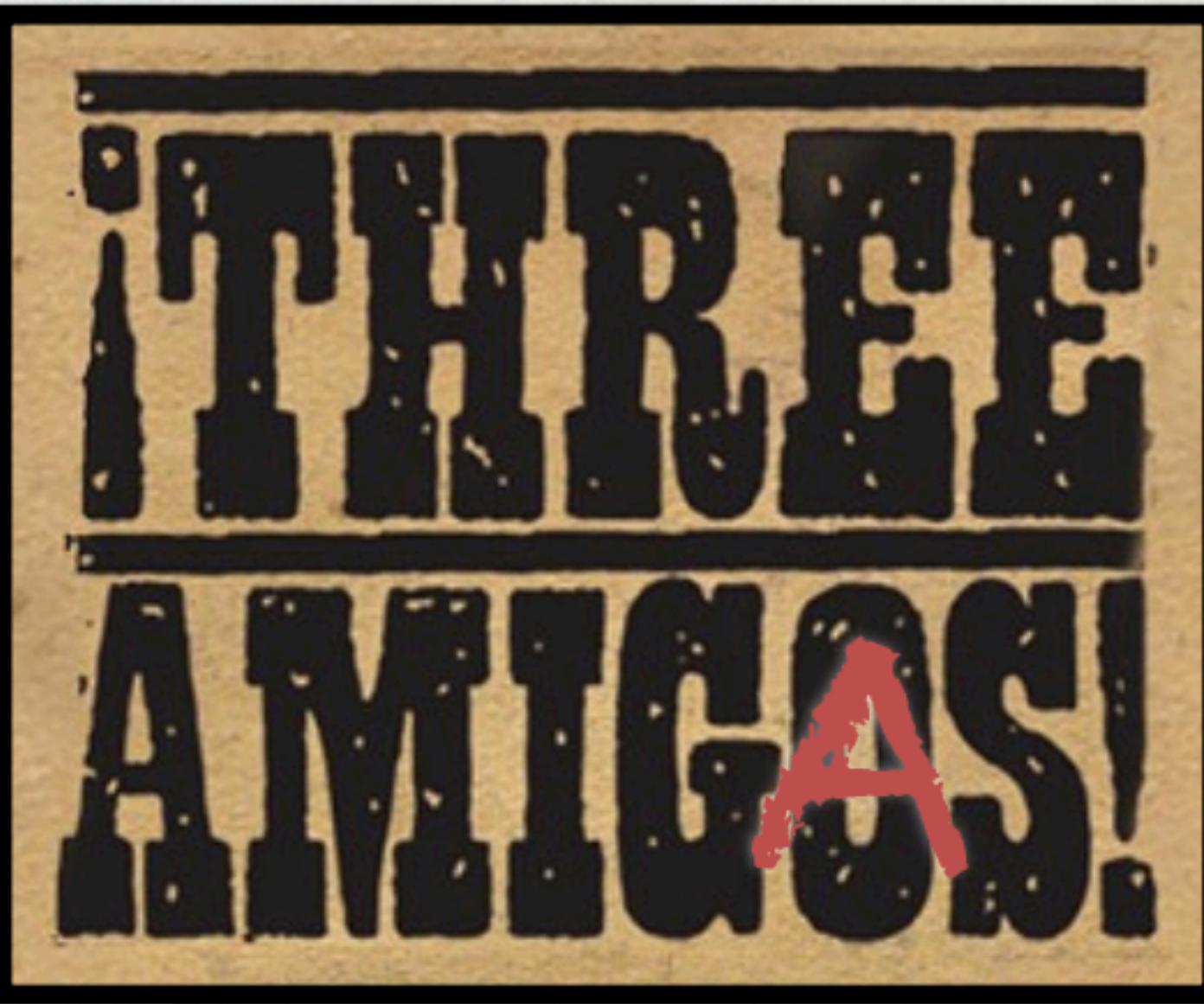
- Domain experts are:
 - Subject-matter experts who know all the intricacies of the business that we are going to model and implement in code.
 - Knowledge authorities in the software's business domain
- Domain experts are not:
 - The analysts gathering the requirements
 - The engineers designing the system

Domain Expert Scope

- Some subject-matter experts will have a detailed understanding of how the entire business domain operates
- Others will specialize in particular subdomains

Ubiquitous Language [Strategic]





Ubiquitous Language

- Cornerstone of Domain Driven Design
- A dictionary of the same language
- A language for describing your business domain
- Represents both the business domain and the domain experts' mental models.

Ubiquitous Language Goals

- Represent the companies terms only
- No technical jargon (e.g. RestFUL, Patterns, Kubernetes)
- Frame the domain experts' understanding and mental model
- Make it easy to understand and eliminate assumptions

Consistency

- Each term of the *Ubiquitous Language* should have only one meaning
- If it doesn't it may fall into one of the following categories:
 - Ambiguous Terms
 - Synonymous Terms

Ambiguous Terms

- Example: In business domain, the term policy has multiple meanings:
 - A regulatory rule
 - An insurance contract
- Ubiquitous language demands a single meaning for each term
- “policy” should be modeled explicitly using the two terms regulatory rule and insurance contract

Synonymous Terms

- Two terms cannot be used interchangeably in a ubiquitous language
- Example: user
- User may be used interchangeably: user, visitor, administrator, account, etc.

Ubiquitous Language in Artifacts

- Ubiquitous Language should be used in
 - Requirements
 - Tests
 - Documentation
 - Source Code
- Of course, it should match the language of the business

Ubiquitous Language in Artifacts

- Ubiquitous Language should be used in
 - Requirements
 - Tests
 - Documentation
 - Source Code
- Of course, it should match the language of the business
- Make it a centerpiece and include a source of truth glossary

Gherkin and BDD

Given the account has a balance of \$23.95

When a message of "balance" comes in from a 404-333-2222

Then a text is sent back: "Your balance is \$23.95"

Given the account has a balance of \$23.95

When any message comes in from an unknown number 233-123-1121

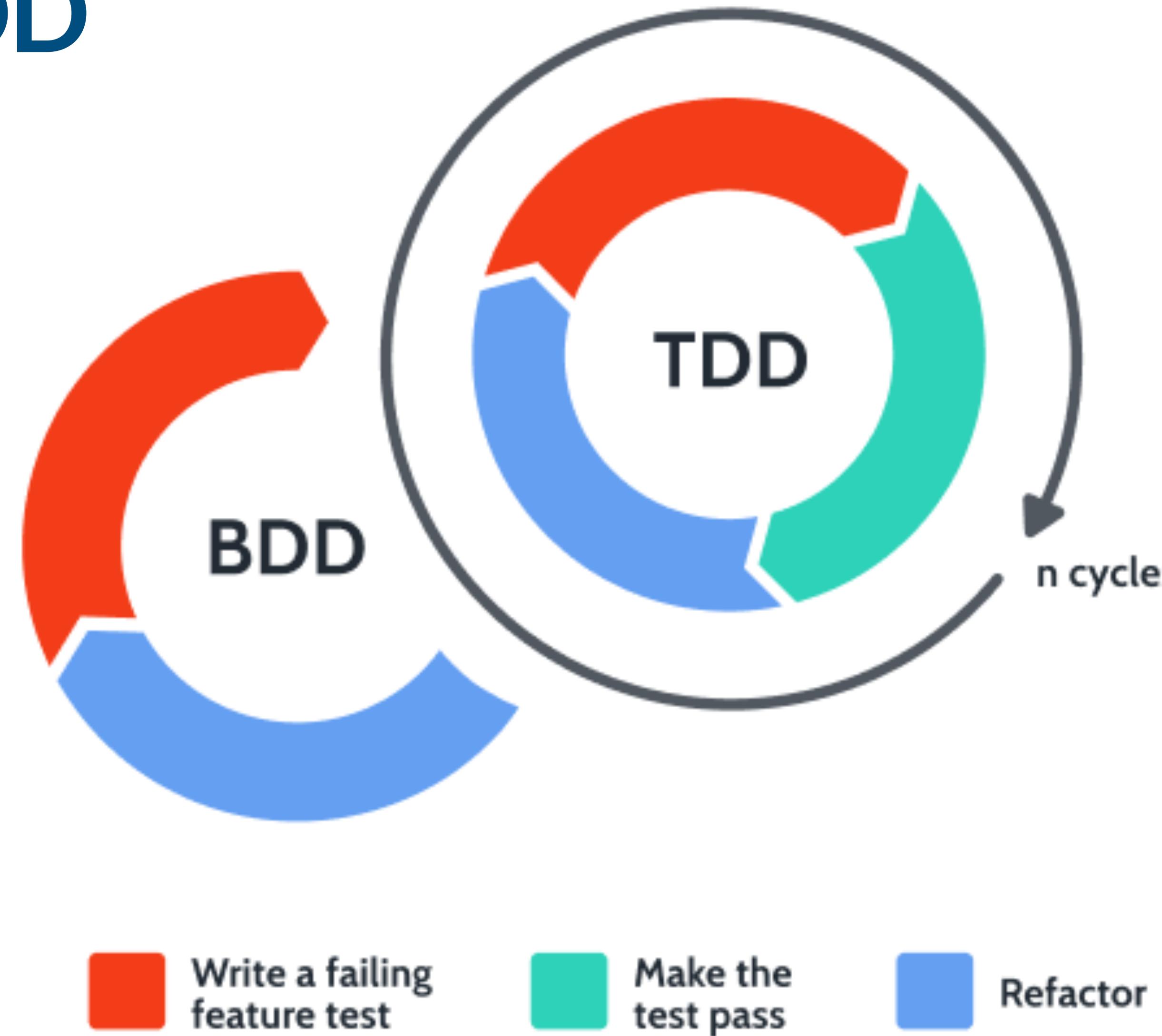
Then log message and phone number, and do nothing

Given the account has a balance of \$23.95

When any message other than "balance" comes in from a 404-333-2222

Then send text back: "Didn't understand message, send BALANCE for current balance."

BDD & TDD



Bounded Contexts [Strategic]



Inconsistent Models

- Ubiquitous Language should be consistent since it will drive decisions
- As companies scale, the domain experts' mental models will be inconsistent
- Domain experts will use synonymous terms and ambiguous terms
- Some will be hard to detect after initial assessment
- Lead, Sale, Account, Candidate, Order - These are few terms that may have different meanings, albeit slight, depending on who you ask

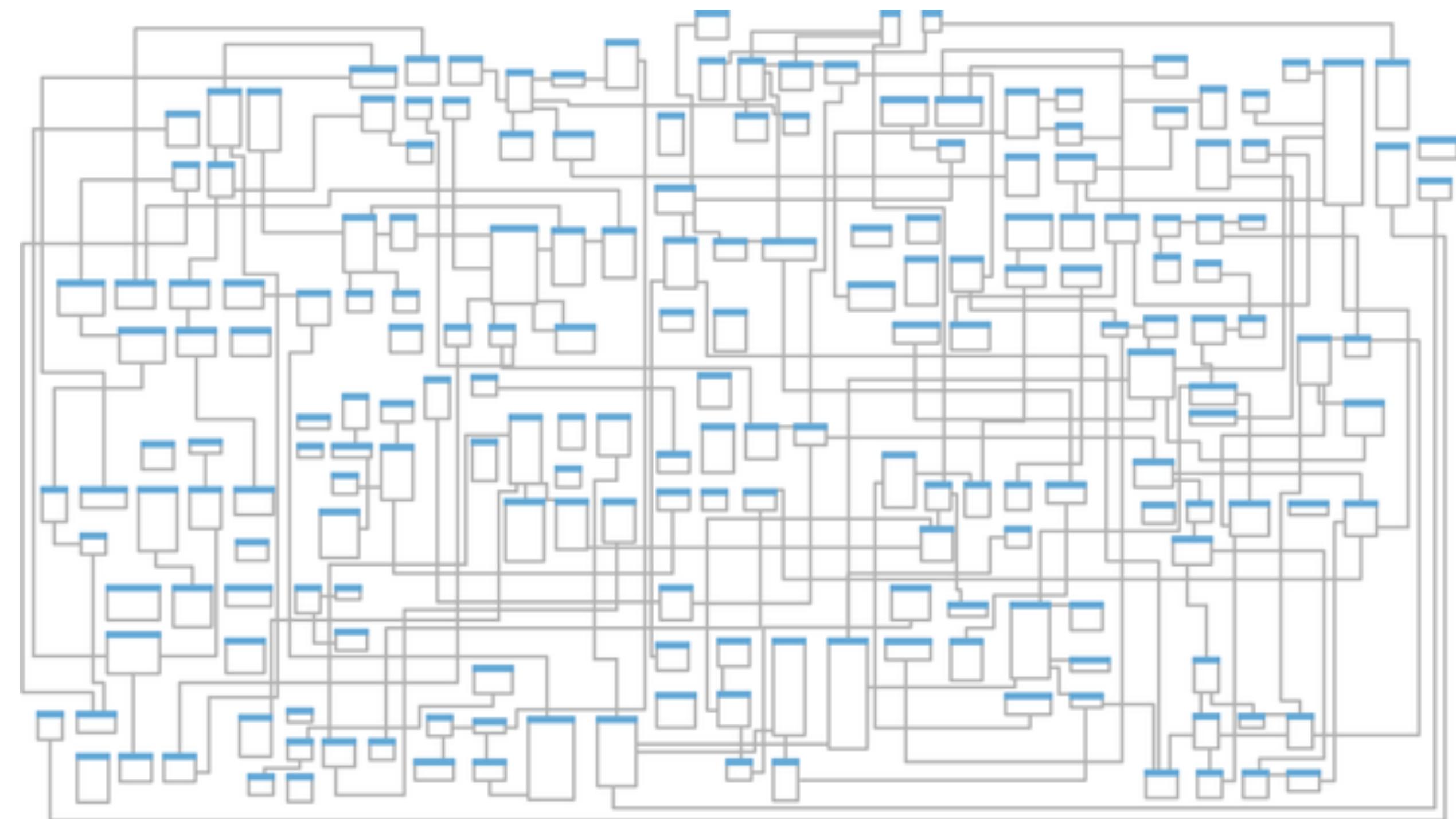


Dealing with Ambiguity

- Ubiquitous Language needs to be consistent - each term should have one meaning
- Ubiquitous Language needs to map to domain experts' mental models
- Having two leads isn't a problem in communication, but it is in software

The Traditional Approach

- The Traditional Approach for such modeling a single model
- Typically defined by an enormous entity relationship (ERD)
- "Suitable for everything but eventually are effective for nothing"

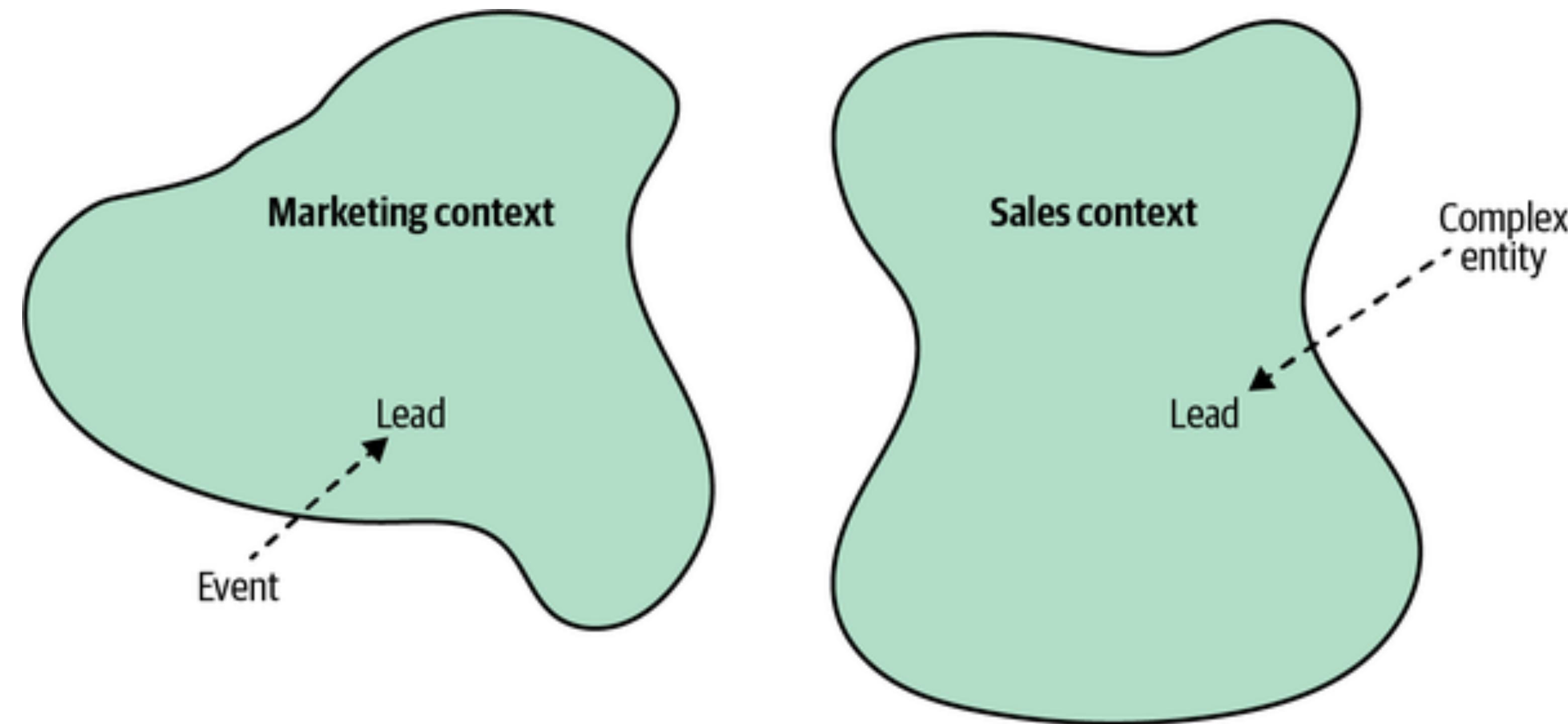


Would Prefixing Work?

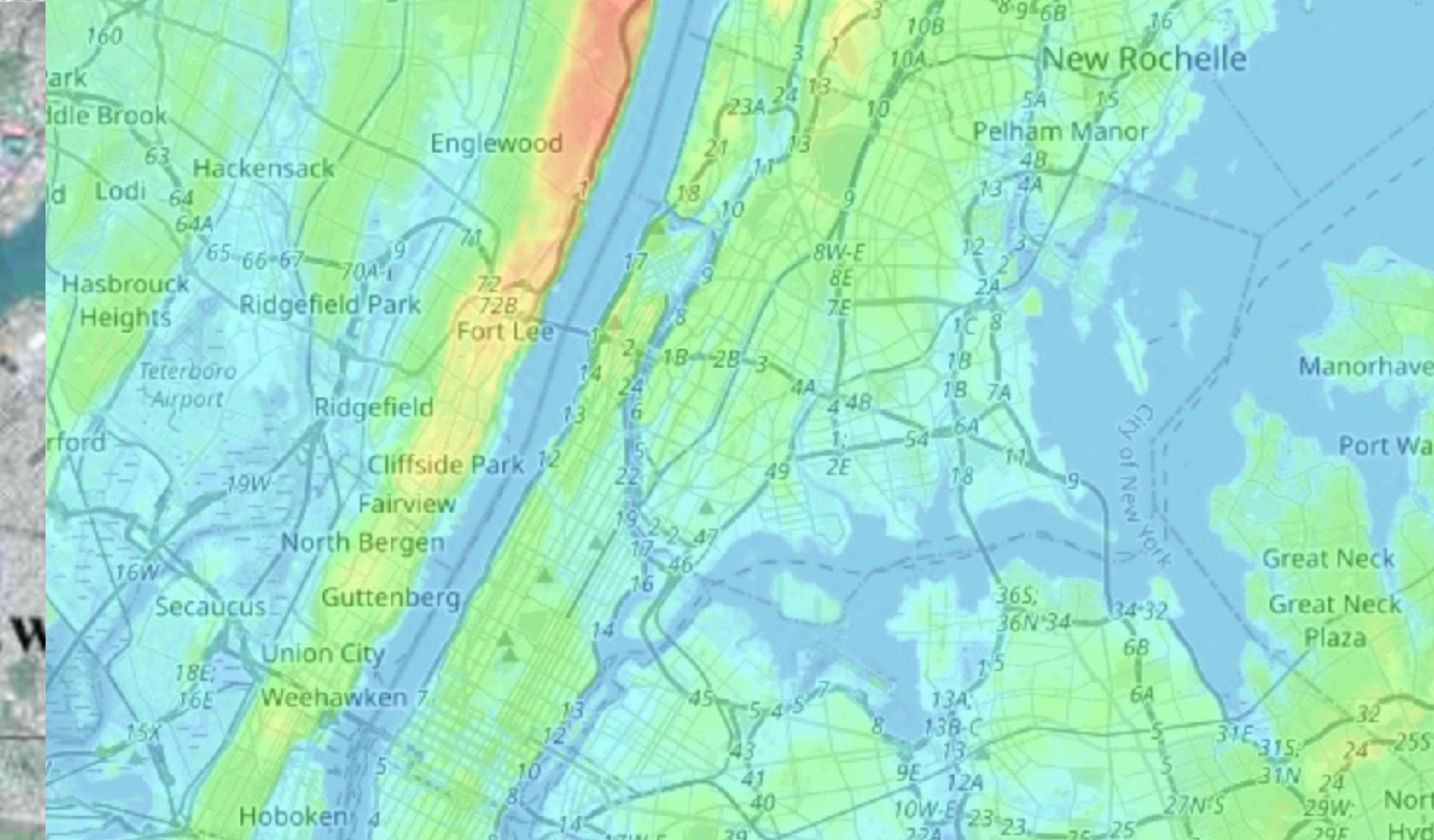
- Can we do “marketing lead” and “sales lead”?
- Allows for two different models of code
- This has three main disadvantages:
 - Creates cognitive load. When should each be used? The closer they are the likely a mistake
 - The implementation will not be aligned with the ubiquitous languages
 - Does anyone use the prefixed terms in everyday language?

Bounded Contexts

- Divide the Ubiquitous Language into multiple smaller languages
- Assign each team to the explicit context in which it can be applied: *its bounded context*



The Bronx



Ubiquitous Language is not universal

- One bounded context can be completely irrelevant to the scope of another
- They allow distinct models according to different problem domains
- Bounded contexts are the consistency boundaries of ubiquitous language

How big should the bounded context be?

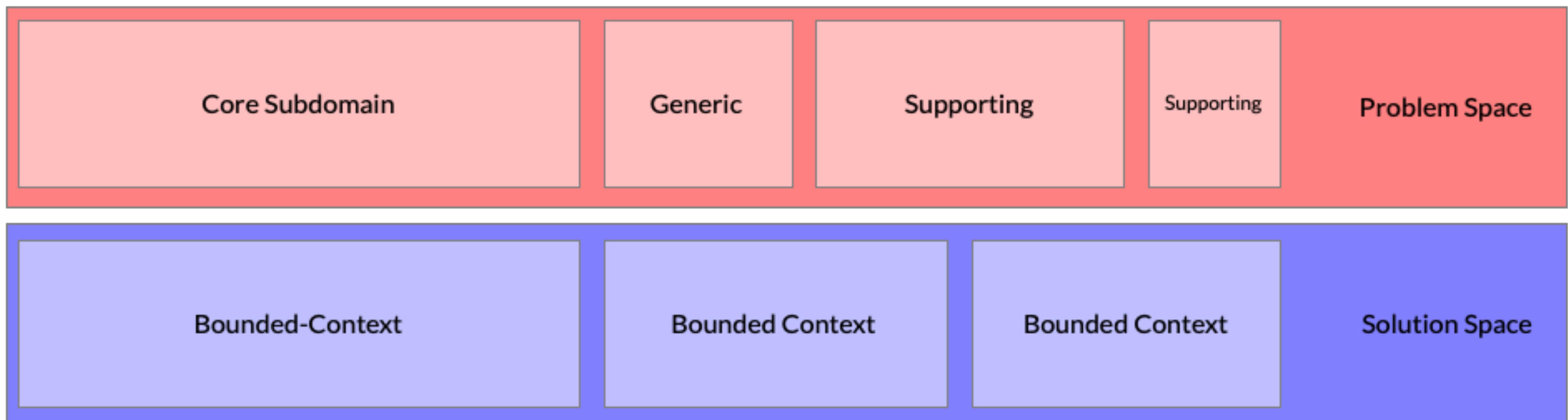
- It should be useful and based on your problem domain
- The wider the boundary of the ubiquitous language is, the harder it is to keep it consistent.
- The smaller the boundary of the ubiquitous language, the more integration overhead the design induces

Bounded Context vs Subdomains

- Subdomains
 - Recall identification of different subdomains: core, supporting, generic
 - Subdomains are interrelated use-cases and defined by the business domain and system requirements
- Bounded Contexts
 - Designed - A strategic designed decision
 - We decide how to divide the business domain into smaller, manageable problem domains

Bounded Context vs Subdomains

- Subdomains are part of the **problem space** - "What is in place and what are we solving"
- Bounded Contexts are part of the **solution space** - "How do we solve the problems"



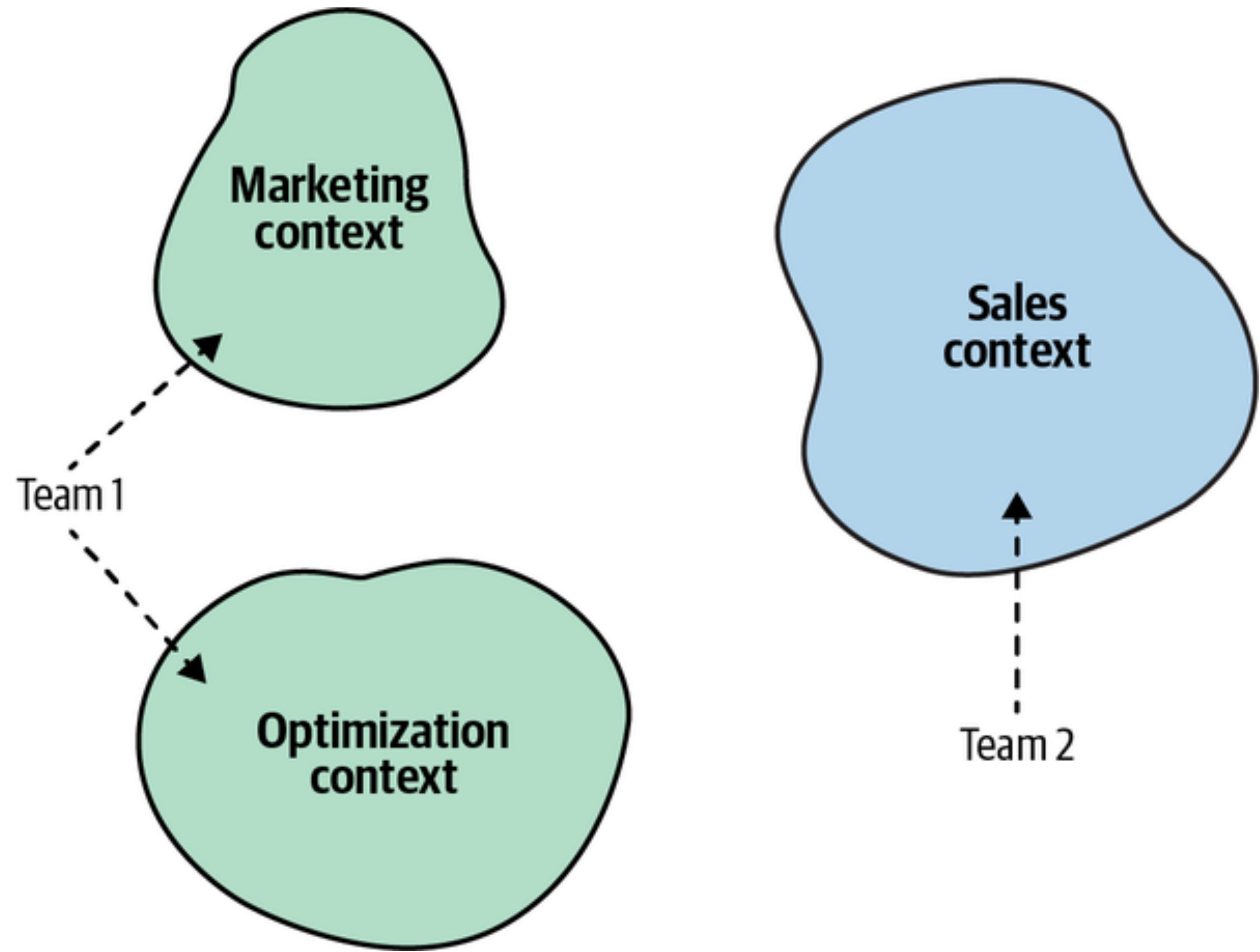
Physical and Ownership Boundaries

Physical Boundary

- Each bounded context should be implemented as an individual service/project
- It is implemented, evolved, and versioned independently of other bounded contexts
- Clear physical boundaries between bounded contexts allow us to implement each bounded context with the technology stack that best fits its needs
- A bounded context can contain multiple subdomains. In such a case, the bounded context is a physical boundary, each of its subdomains is a logical boundary
- Logical boundaries bear different names in different programming languages: namespaces, modules, or packages.

Ownership Boundary

- A bounded context should be implemented, evolved, and maintained by one team only
- No two teams can work on the same bounded context
- Segregation eliminates implicit assumptions that teams might make about one another's models



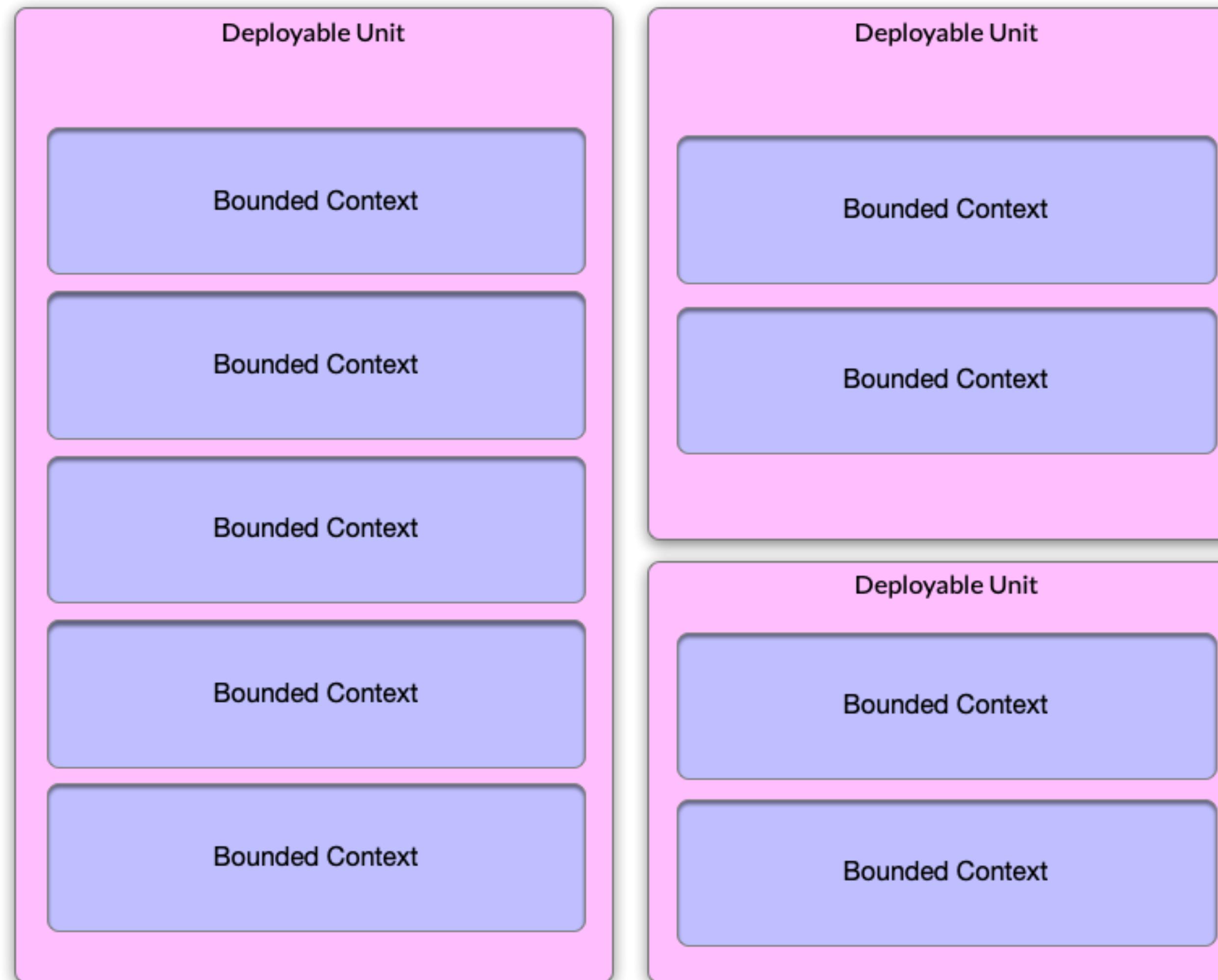
Microservices and Bounded Contexts

- Controversial Topic. Yes, it is. Take some time to look at varying opinions.
- Microservices came out in 2011, Bounded Contexts came out in 2003.
- But they are a compliment. Remember the rules in the previous slide on physical and ownership boundaries

Microservices and Bounded Contexts

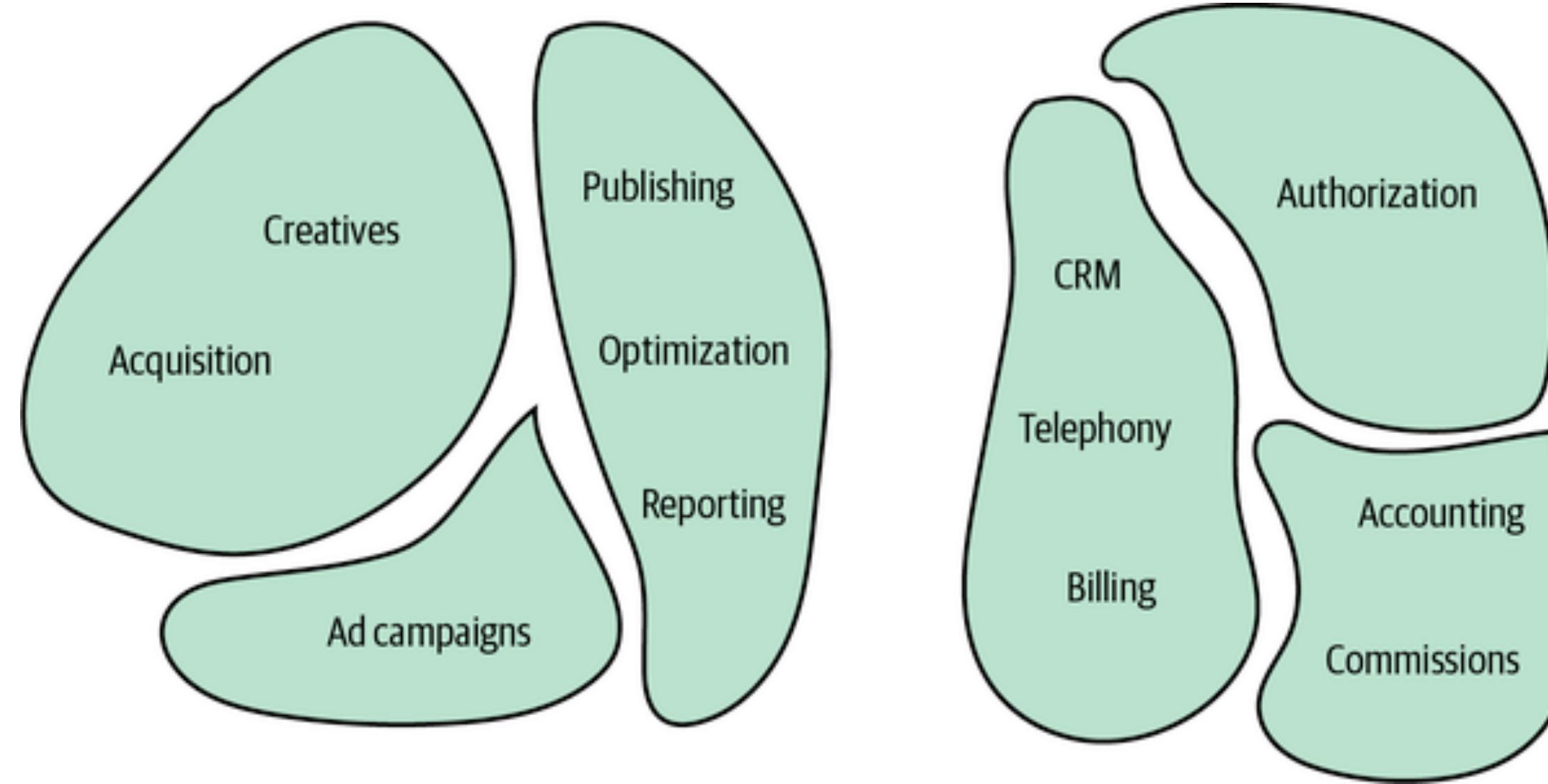
- **A Bounded Context is a Conceptual Boundary:**
 - It ensures that a specific domain model has clear meaning within its boundary.
 - It's about creating language and modeling consistency.
- **A Microservice is a Deployment Unit:**
 - It's a technical implementation detail that allows independent deployment, scaling, and evolution.

Microservices and Bounded Contexts



The Diagram

Contexts are neither big nor small, but useful



Integrating Bounded Contexts

[Strategic]



Contracts

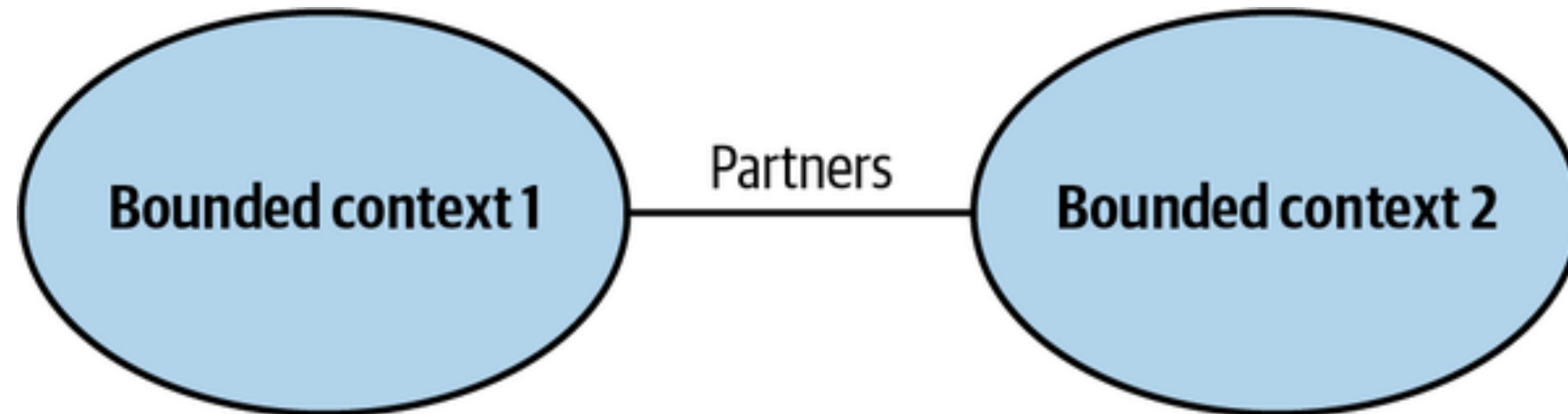
- A system cannot be built out of independent components, the components have to interact with one another to achieve the system's overarching goals
- The same goes for implementations in bounded contexts.
- Although they can evolve independently, they have to integrate with one another
- These are called *contracts*

Contracts Between Contracts

- By definition, two bounded contexts are using different ubiquitous languages.
- Which language will be used for integration purposes?
- These integration concerns should be evaluated and addressed

Partnership

- Integration between bounded contexts is coordinated in an ad hoc manner
- One team can notify a second team about a change in the API, and the second team will cooperate and adapt—no drama or conflicts



Partnership Coordination

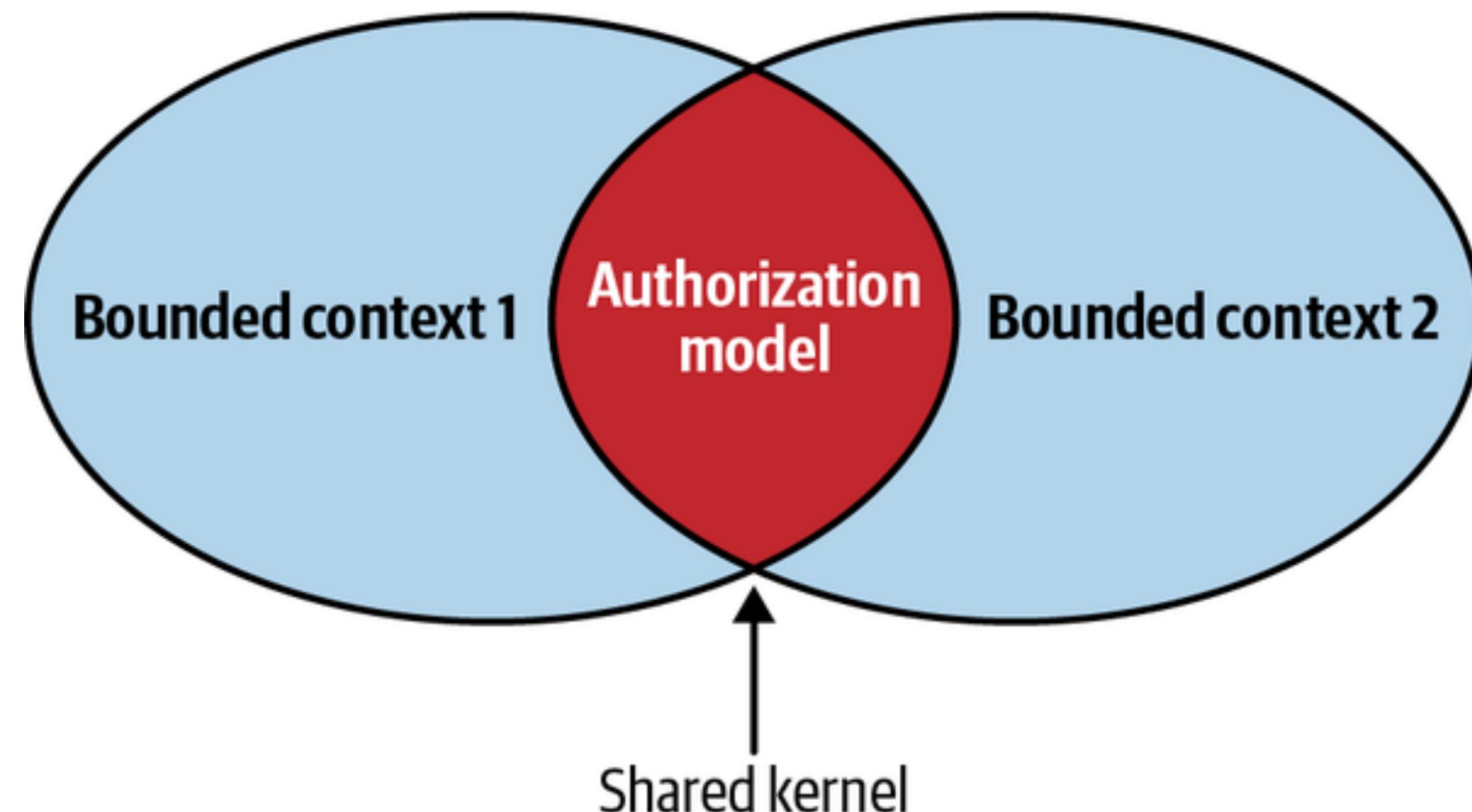
- Coordination is two-way
- No one team dictates the language that is used for defining the contracts
- Teams work out the differences and choose the most appropriate solution
- Both cooperate, and neither is interest in blocking another

Partnership Requirements

- Commitment
- Constant Communication
- Continuous Integration to minimize the integration feedback loop
- Both teams should highly likely be geographically in the same location

Shared Kernel/Model

- Same model of a subdomain, or a part of it, implemented in multiple bounded contexts
- Designed according to the needs of all of the bounded contexts
- Consistent across all of the bounded contexts that are using it

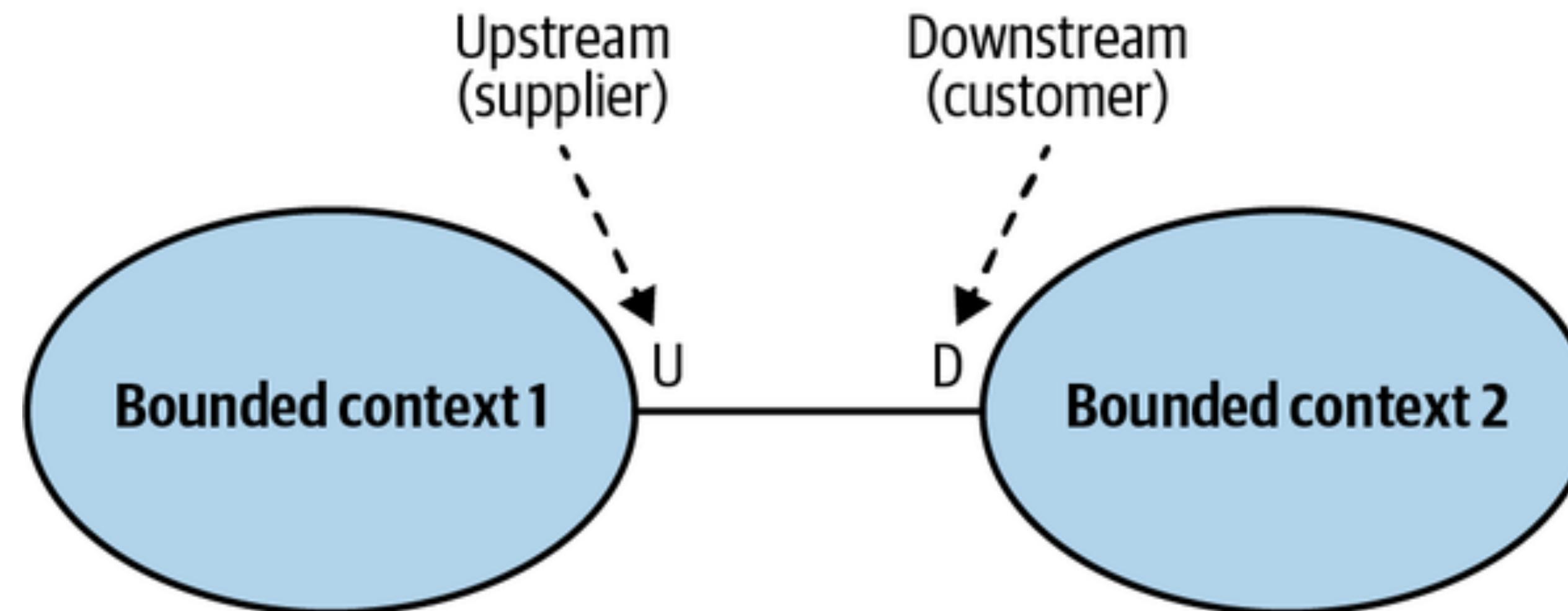


Shared Kernel/Model

- Example: Custom Authentication/Authorization module used by separate bounded contexts
- Implemented as a mono-repo - sharing the same files per one repository
- Implemented as a dedicated-repo - dedicated project and referenced as a library
- Tight CI/CD
- *Using a shared kernel violates the independence rule of bounded context, and therefore, should be justified*
- *It can be used as migratory tool to either combine bounded contexts or split them apart*

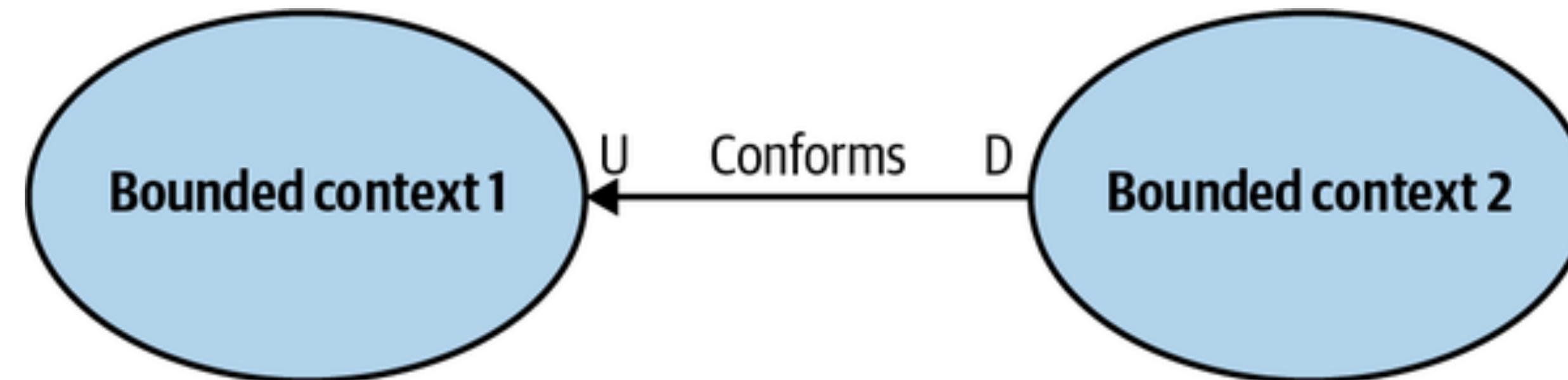
Customer-Supplier

- The service provider is “upstream” and the customer or consumer is “downstream.”
- Both teams (upstream and downstream) can succeed independently
- Imbalance of power: either the upstream or the downstream team can dictate the integration contract
- This is where a decision should be made about the relationship



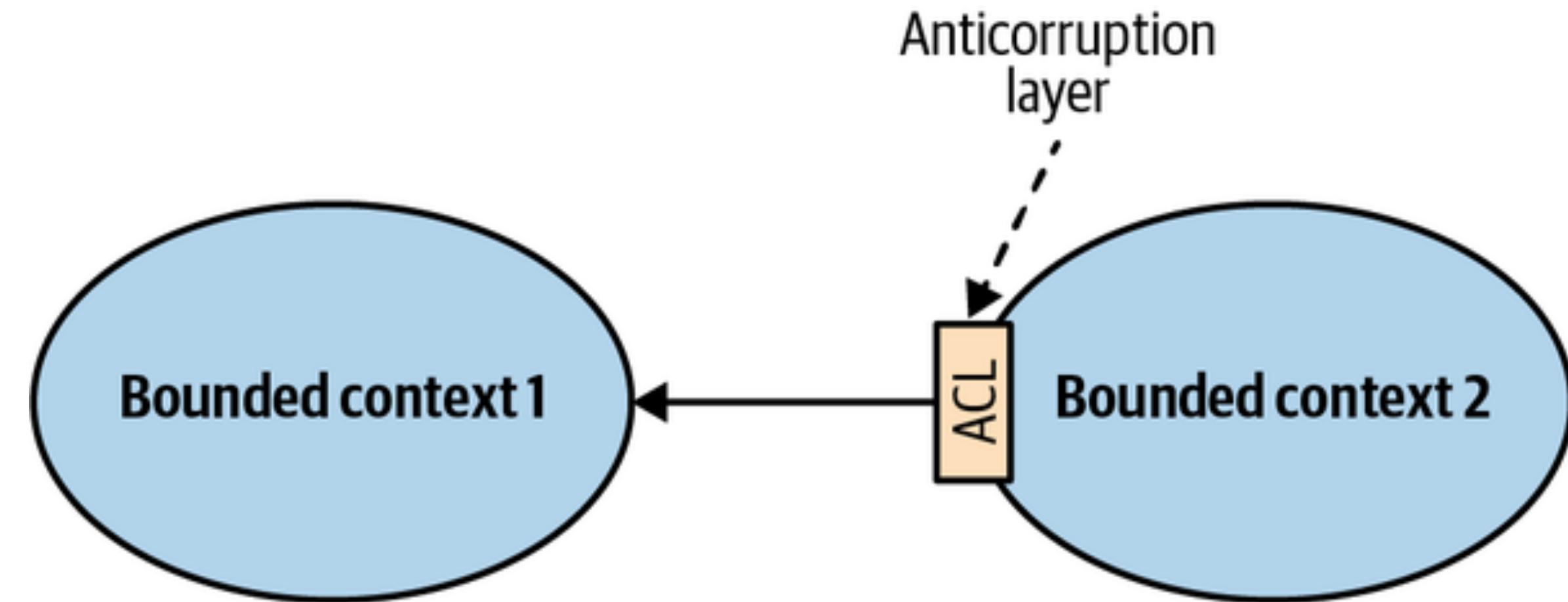
Conformist

- Balance of power favors the upstream team, no real motivation to support the client
- Upstream provides the integration contract, defined according to its own model—take it or leave it
- Caused by integration with service providers that are external to the organization or simply by organizational politics
- The contract exposed by the upstream team may be an industry-standard, well-established model, or it may just be good enough for the downstream team's needs.



Anti-Corruption Layer

- The balance of power in this relationship is still skewed toward the upstream service.
- In this case, the downstream bounded context is not willing to conform.
- Instead, it translates the upstream bounded context's model into a model tailored to its own needs via an anticorruption layer

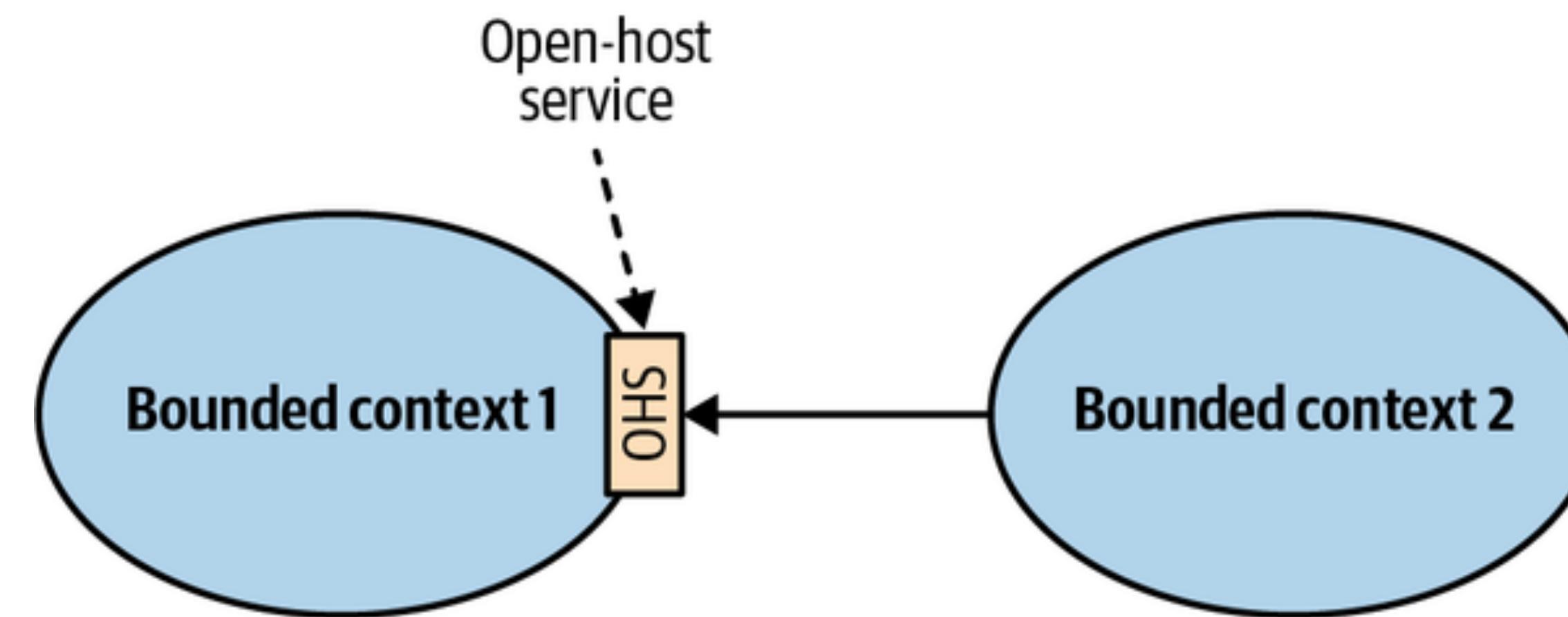


Reasons for Anti-Corruption Layer

- A core subdomain's model requires extra attention, and adhering to the supplier's model might impede the modeling of the problem domain.
- When the upstream model is inefficient or inconvenient for the consumer's needs
- If a bounded context conforms to a mess, it risks becoming a mess itself (legacy systems)
- When the supplier's contract changes often, the consumer wants to protect its model from frequent changes

Open Host Service

- Power is toward the consumers
- Supplier is interested in protecting its consumers and providing the best service possible.
- Upstream supplier decouples the implementation model from the public interface.
- Allows the supplier to evolve its implementation and public models at different rates
- The supplier exposes a protocol convenient for the consumers

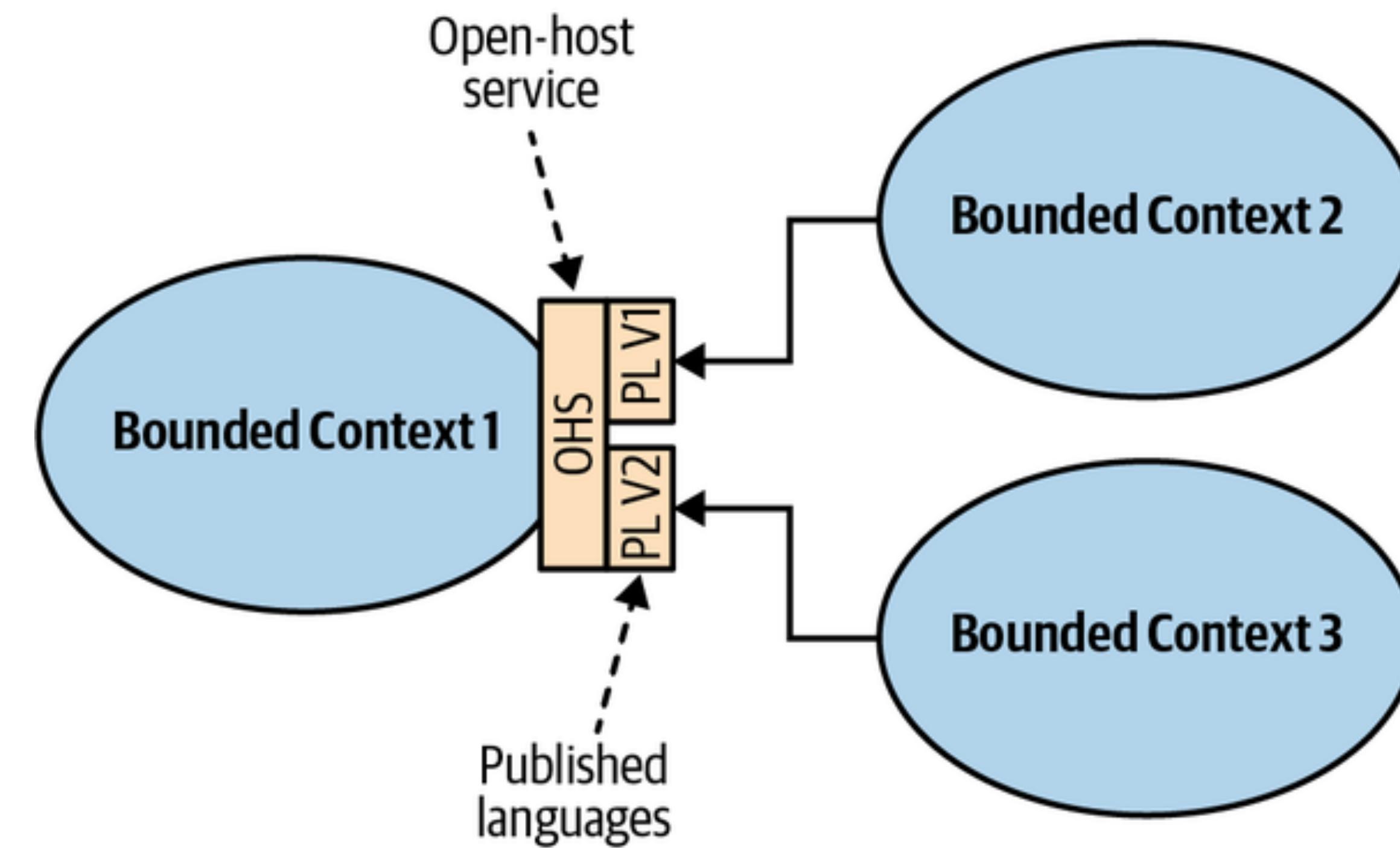


Published Language

- Publicly used to communicate and integrate between different Bounded Contexts.
- They also have an agreed-meaning across Bounded Contexts.
- Mainly used by engineers working in different Bounded Contexts to agree on integration approach (open-host, client-server, pub-sub, etc.).
- Mainly expressed via technical format (API, JSON, XML, Protocol Buffer, etc.)

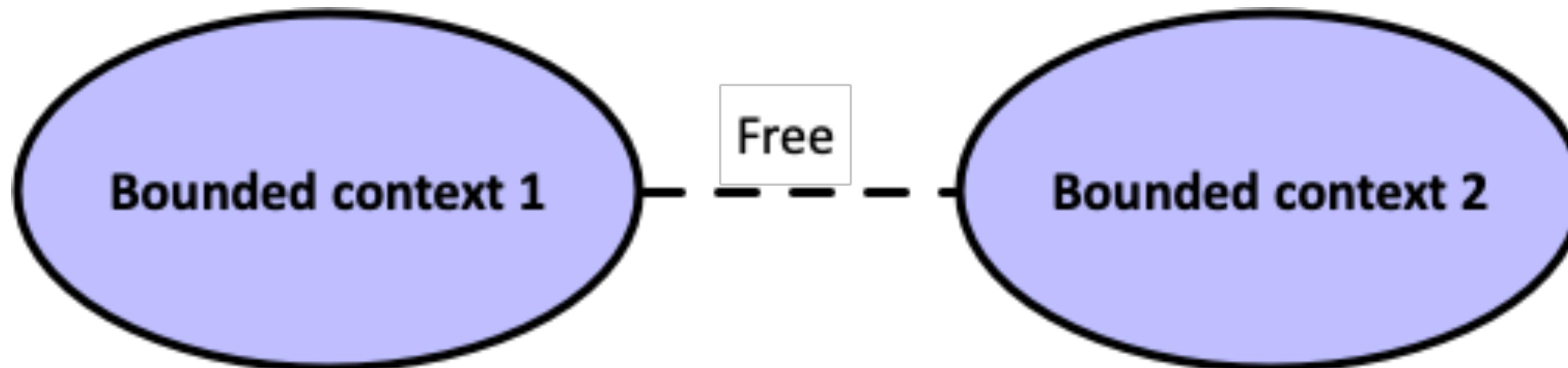
Open Host Service Multiple Protocols

- Integration model's decoupling allows the upstream bounded context to simultaneously expose multiple versions of the published language
- Allowing the consumer to migrate to the new version gradually



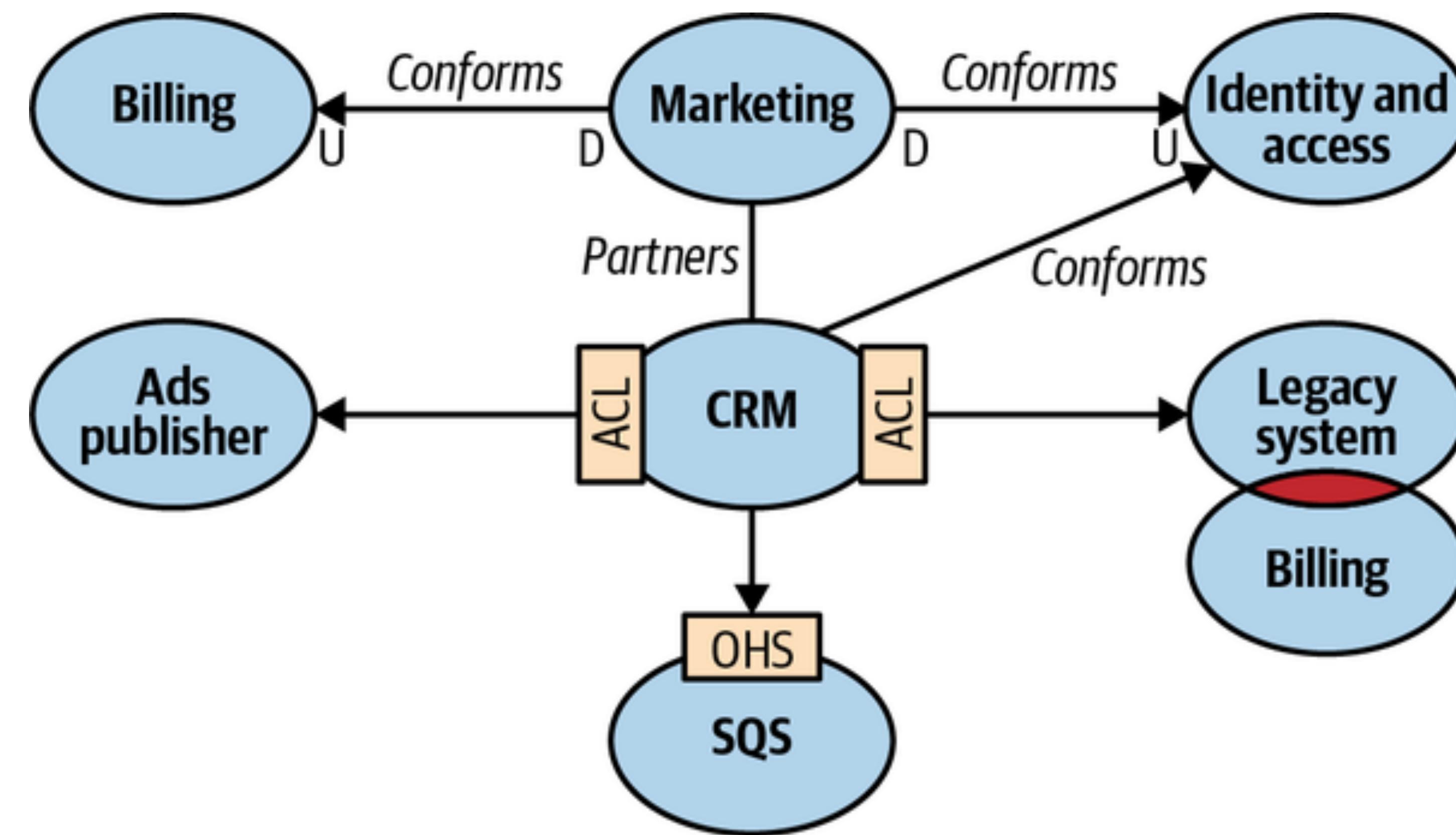
Separate Ways/Free

- Last option or pattern is not to collaborate at all for the following reasons:
 - **Communication Issues** - Organization Size and Politics hinders collaboration
 - **Generic Subdomains** - If the generic solution is easy to integrate, it may be more cost-effective to integrate it locally in each bounded context.
 - **Model Differences** - The models may be so different that a conformist relationship is impossible. Implementing an anticorruption layer would be more expensive than duplicating the functionality



Creating a Context Map

Context map is a visual representation of the system's bounded contexts and the integrations between them





Documentation

Project Background

Getting Involved

News

Search...



A Modeling Framework for Strategic Domain-driven Design

ContextMapper is an open source project providing a Domain-specific Language and Tools for Strategic Domain-driven Design (DDD), Context Mapping, Bounded Context Modeling, and Service Decomposition.

Quick Start

- [What is Context Mapper?](#) Learn more about the project's background.
- New to Context Mapper? [Get Started](#).
- Installation Links: [Eclipse Plugin](#), [Visual Studio Code Extension](#), [Online IDE](#)
- [Example Models](#)
- [Latest News and Release Notes](#)
- Learn more about: [Context Mapping DSL \(CML\)](#), [Architectural Refactorings](#), [Generators](#), [Rapid OOAD](#), [Service Decomposition](#), [Reverse Engineering](#), [Microservice Generation with JHipster](#), [Event Storming](#), [CQRS](#)



**CONTEXT
MAPPER**

<https://contextmapper.org/>

Demo: Context Mapper



- Let's see briefly what a context mapper looks like.

Simple Applications [Tactical]

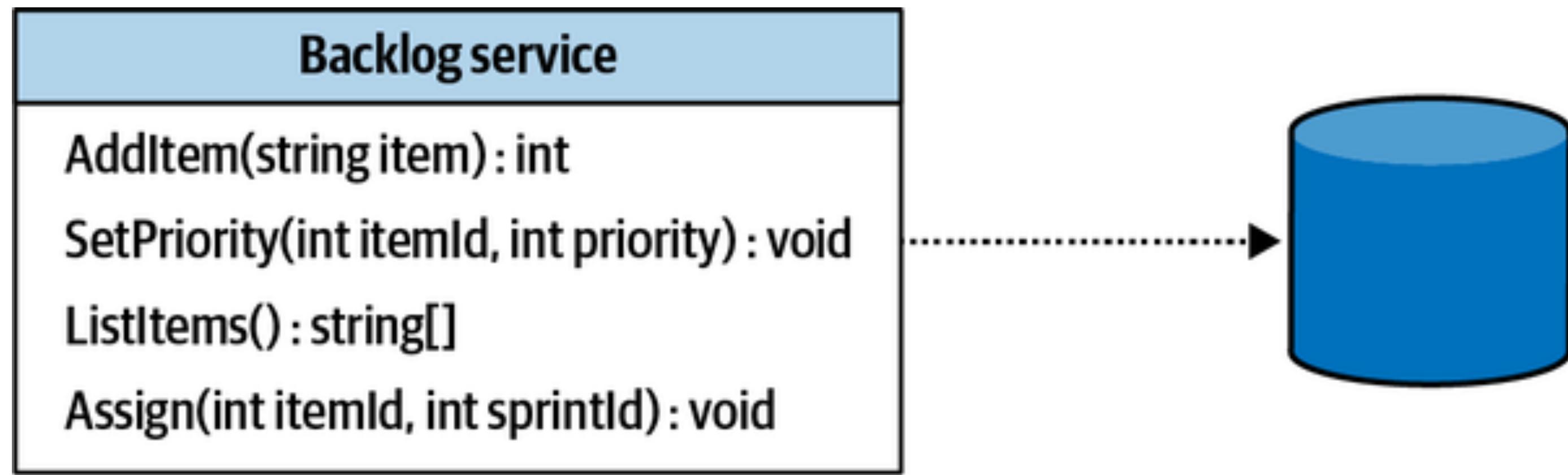


Simple Business Logic

- For supporting subdomains we can use either one of these software architectures:
 - Transaction Script
 - Active Record

Transaction Script

- Organizes business logic by procedures where each procedure handles a single request from the presentation
- A system's public interface can be seen as a collection of business transactions that consumers can execute
- Transactions can retrieve information managed by the system, modify it, or both
- Pattern organizes the system's business logic based on procedures
- Each procedure implements an operation that is executed by the system's consumer via its public interface
- Public operations are used as encapsulation boundaries



Demo: Transaction Scripts

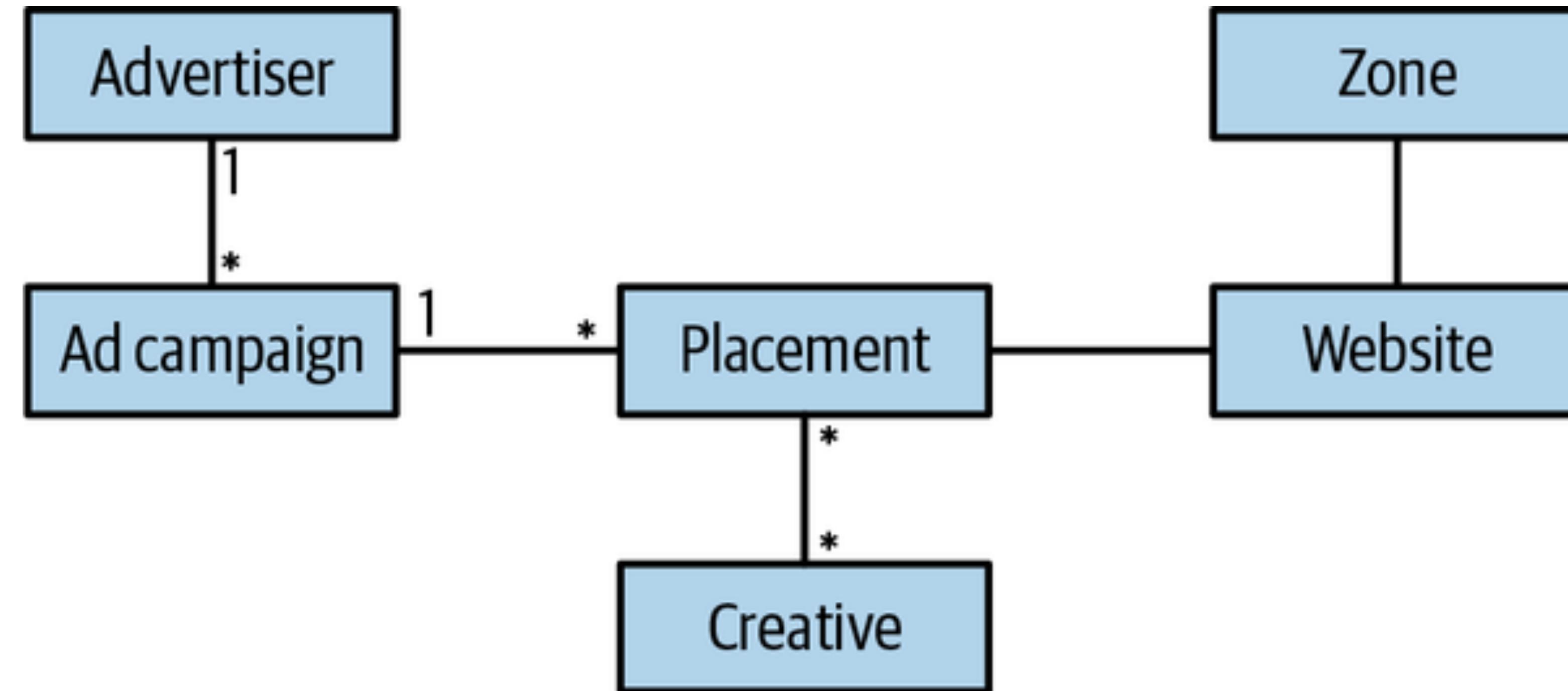


- Let's see an example of a transaction script

Active Record

- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data
- Active record supports cases where the business logic is simple
- Business logic may operate on more complex data structures which is different than transaction script
- Active record also supports complicated trees and hierarchies
- Doing something like the following diagram would make it so complicated for something like a transaction script which leads to repetitive code

Example of a Complex Active Record Structure



Implementation of an Active Record

- Active Record uses dedicated objects, known as active records, to represent complicated data structures
- Active Record objects also implement data access methods for creating, reading, updating, and deleting records—the so-called CRUD operations
- Active record objects are coupled to an object-relational mapping (ORM) or some other data access framework

Advantages and Disadvantages to an Active Record

- **Advantage**
 - Active record is essentially a transaction script that optimizes access to databases
 - Can only support relatively simple business logic, such as CRUD operations, which, at most, validate the user's input
- **Disadvantage**
 - It can potentially introduce more harm than good when applied in the wrong context
 - It can be considered an anti-pattern, since it mixes two concerns, domain and repository

Demo: Active Records



- Let's see an example of an active record

Complex Applications [Tactical]



Domain Model Pattern

- The domain model pattern is intended to cope with cases of complex business logic.
- Instead of CRUD, we deal with complex state transitions, business rules, and invariants

Implementation of the Domain Model Pattern

- A domain model is an object model of the domain that incorporates both behavior and data
- DDD's tactical patterns – aggregates, value objects, domain events, and domain services— are the building blocks of such an object model.
- All of these patterns share a common theme: they put the business logic first
- They will all be implemented in plain objects
 - No Infrastructure and no technology concerns like frameworks
 - Adhere to all the correct terms: **Ubiquitous Language**

What goes in the Domain Model Pattern?

- Developers will develop the following software artifacts:
 - Value Objects
 - Entities
 - Aggregates
 - Domain Events
 - Domain Services
 - Application Services

Value Objects

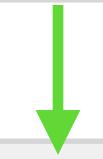
- Object that can be identified by the composition of its values
- No explicit identification is required
- Changing the attributes of any of the fields yields a different object
- Value Objects can be used to counter the “Primitive Obsession” code smell
- Typically immutable

```
class Color {  
    int red;  
    int green;  
    int blue;  
}
```

Entities

- Opposite of a value object and requires explicit identification
- Explicit identification is required
- For example, an Employee, just identified by an employee's first name and last name
- Entities are subject to change

```
class Employee {  
    private Name name;  
    //Constructors,  
    // equals, hashCode, etc.  
}
```



```
class Employee {  
    private EmployeeId employeeId;  
    private Name name;  
    //Constructors,  
    // equals, hashCode, etc.  
}
```

Aggregates

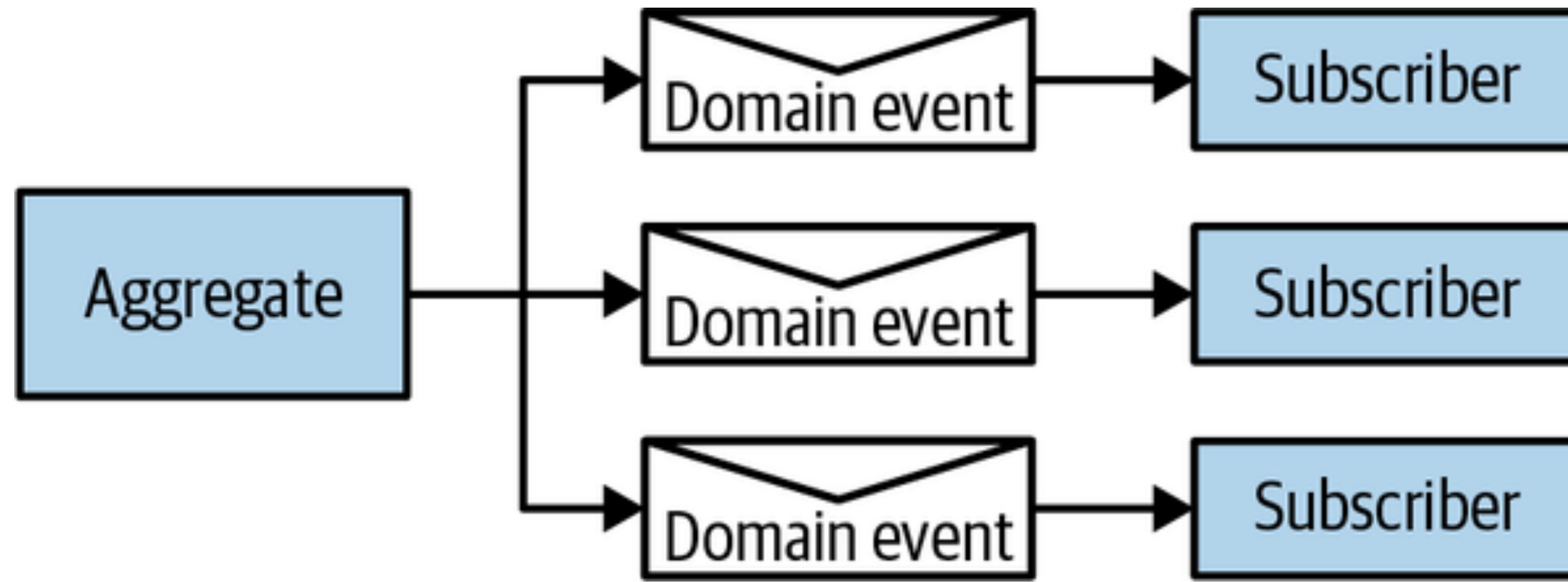
- An entity that manages a cluster of domain objects as a single unit
- Consider an Order to OrderLineItem relationship
- The root entity ensures the integrity of its parts
- Transactions should not cross aggregate boundaries
- They are domain objects(order, playlist), they are not collections, like List, Set, Map

```
class Album {  
    private Name name;  
    private List<Track> tracks;  
  
    public void addTrack(Track track) {  
        this.tracks.add(track);  
    }  
  
    public List<Track> getTracks() {  
        return Collections.copy(tracks)  
    }  
}
```

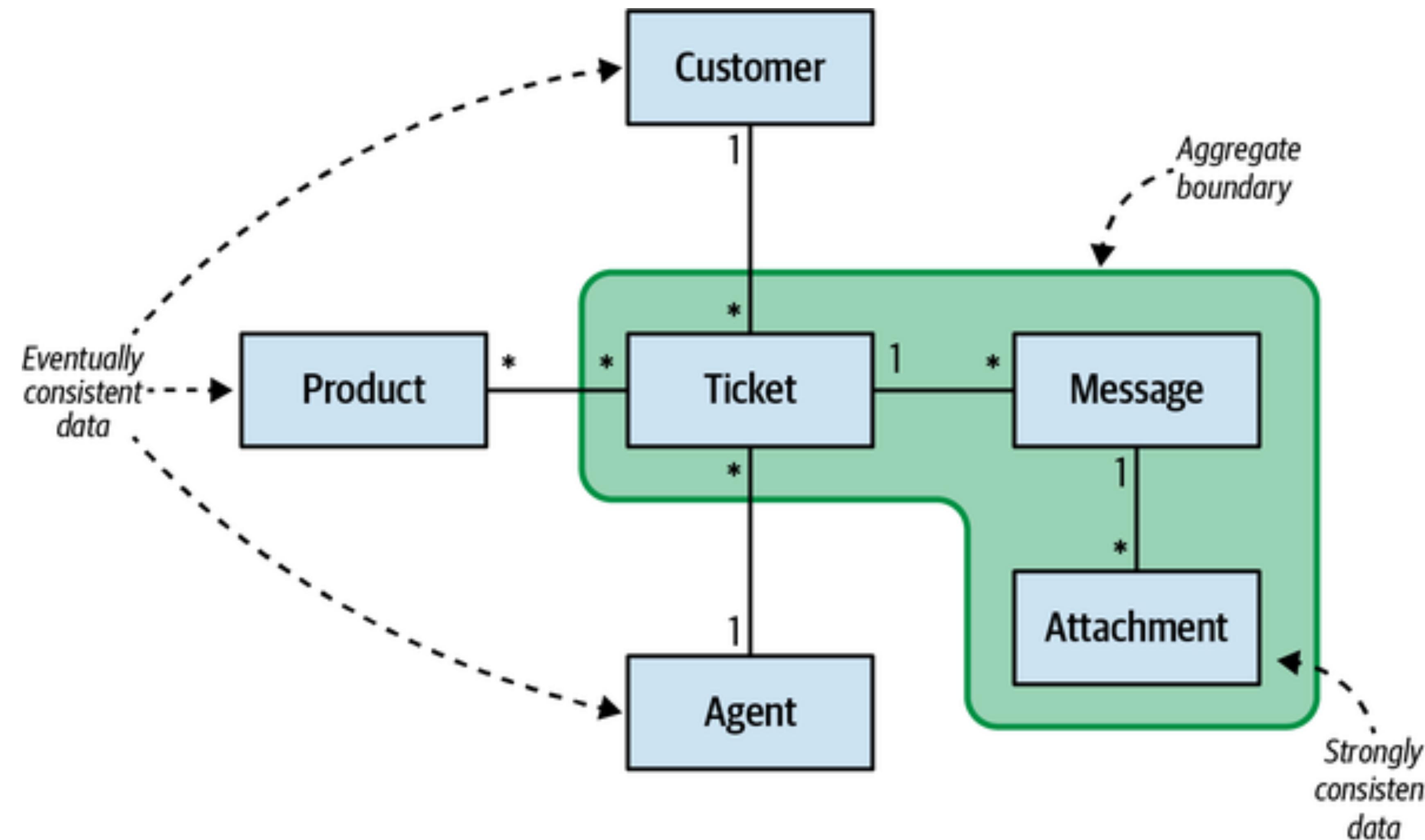
Domain Events

- The identified domain events directly become domain events in the implementation.
- These are immutable objects that capture significant occurrences in the business domain, reflecting changes in state or triggering further actions within the system.
- The events can then emitted from the aggregates and perform other functions

```
public class Ticket {  
    private List<DomainEvent> domainEvents;  
  
    public void execute(RequestEscalation cmd) {  
        if (!this.isEscalated &&  
            this.remainingTimePercentage <= 0) {  
            this.isEscalated = true;  
            var escalatedEvent =  
                new TicketEscalated(id, cmd.Reason);  
            domainEvents.append(escalatedEvent);  
        }  
    }  
}
```



Aggregate as a Consistency Boundary



Keep Aggregates Small

- Keep the aggregates as small as possible and include only objects that are required to be in a strongly consistent state by the aggregate's business logic
- In the following code example, notice that we are not interested in the entire Product nor the entire User, just their identifiers (entities)

```
public class Ticket {  
    private UserId customer;  
    private List<ProductId> products;  
    private UserId assignedAgent;  
    private List<Message> messages;  
}
```

- Reasoning behind referencing external aggregates by ID is to reify that these objects do not belong to the aggregate's boundary, and to ensure that each aggregate has its own transactional boundary.

Domain Services

- Stateless object that implements the business logic
- It naturally doesn't belong to any of the domain model's aggregates or value objects

```
class OrderTax {  
    //no state  
    private void calculateTaxWithRate(Order  
        order, TaxRate taxRate) {  
        //...  
    }  
}
```

Application Services

- Makes use of Repositories (Storage Abstractions)
- Aggregate instances and then sent for transforming to a transformed object
- The transformed object will typically be routed to the UI

```
class OrderService {  
    private OrderRepository orderRepository;  
  
    private void persistOrder(Order) {  
        //...  
    }  
}
```

Demo: Domain Model



- Let's view a Domain Model with Entities, Value Objects, Application Services, etc.

Event Storming [Tactical]

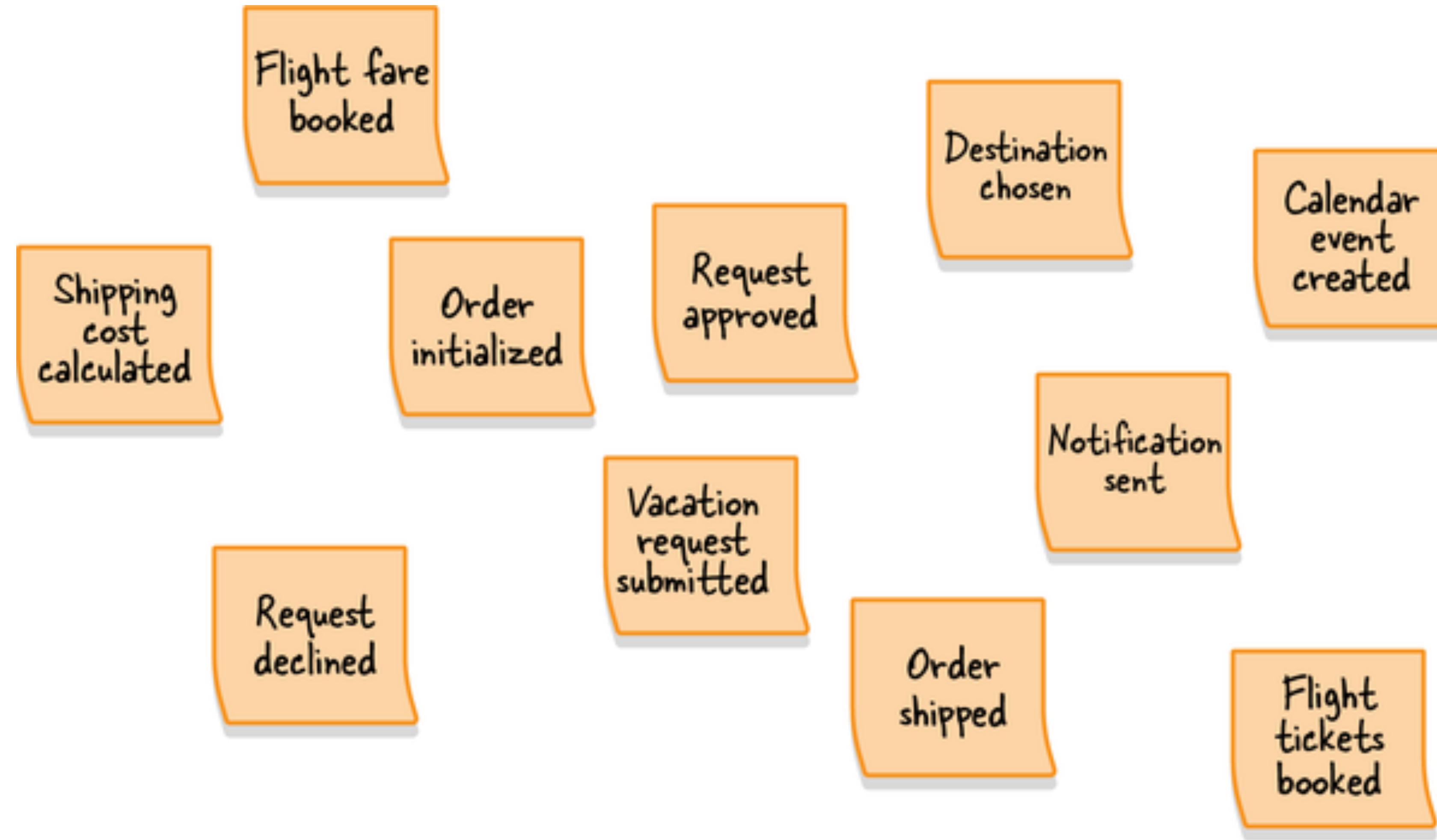


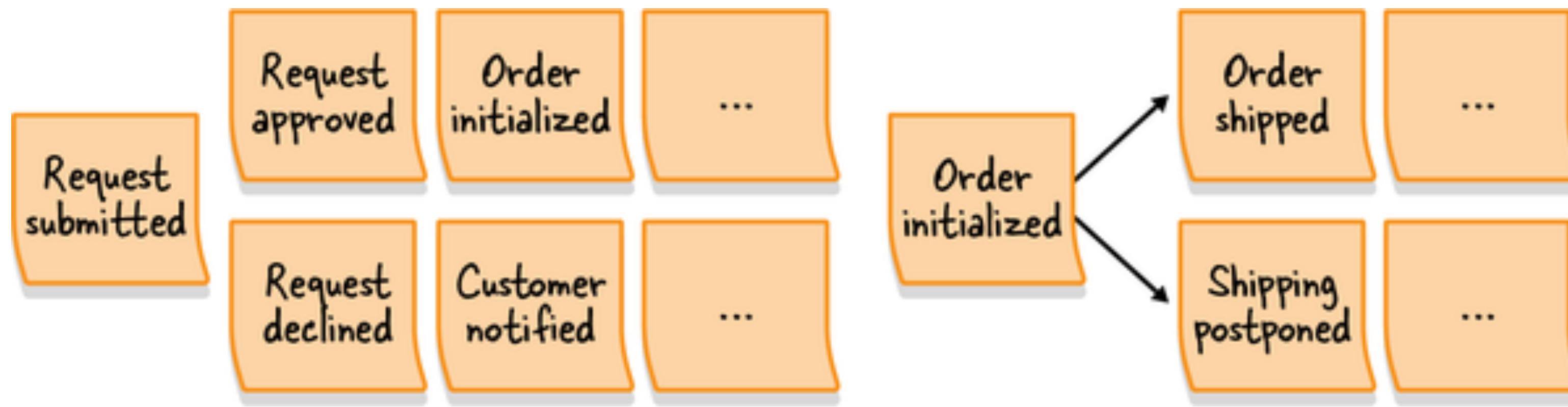
Event Storming

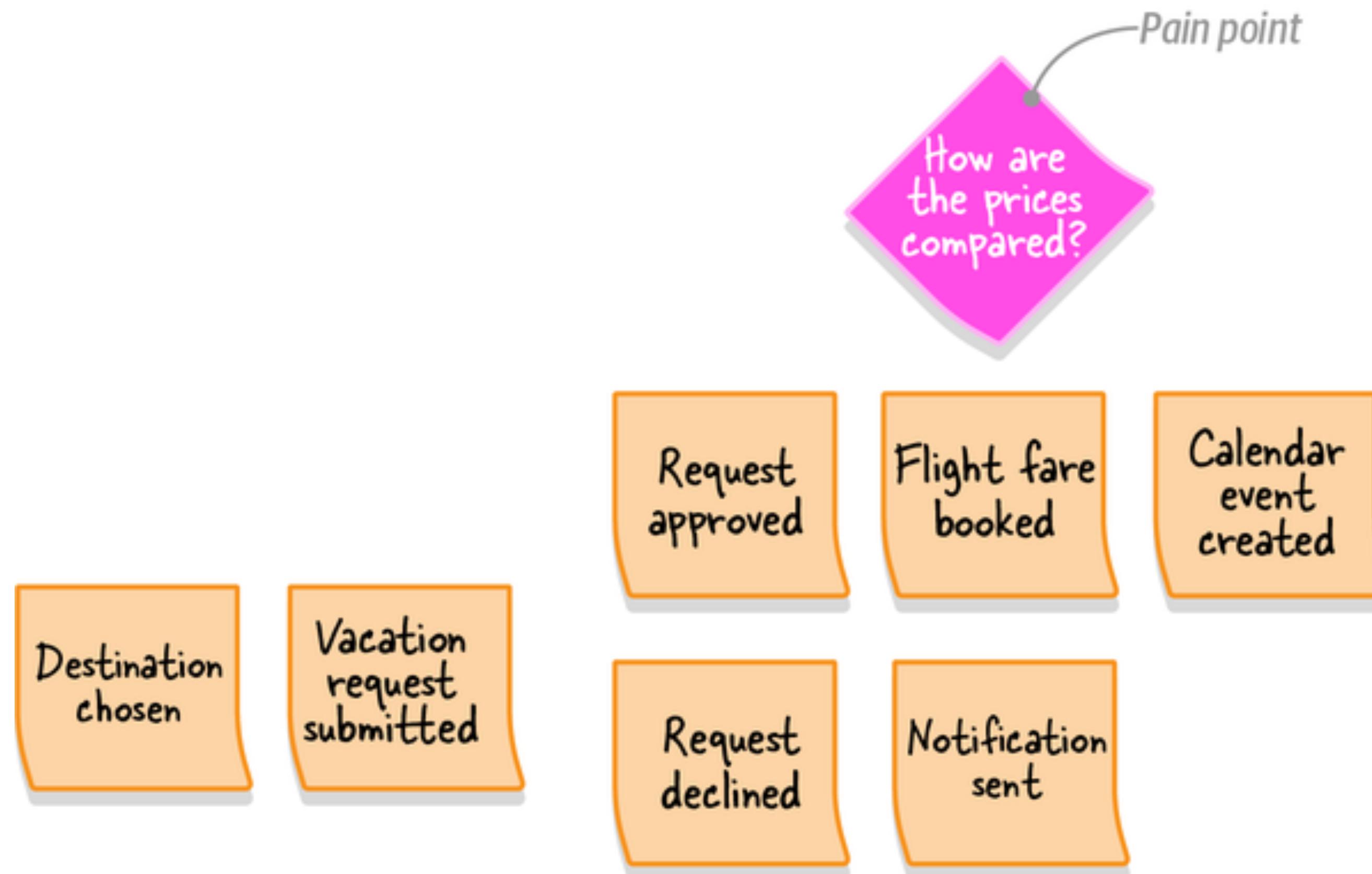
- Low-tech activity for a group of people to brainstorm and rapidly model a business process.
- In a sense, EventStorming is a tactical tool for sharing business domain knowledge.
- An EventStorming session has a scope: the business process that the group is interested in exploring.
- The participants are exploring the process as a series of domain events, represented by sticky notes, over a timeline. Step by step, the model is enhanced with additional concepts—actors, commands, external systems, and others—until all of its elements tell the story of how the business process works.

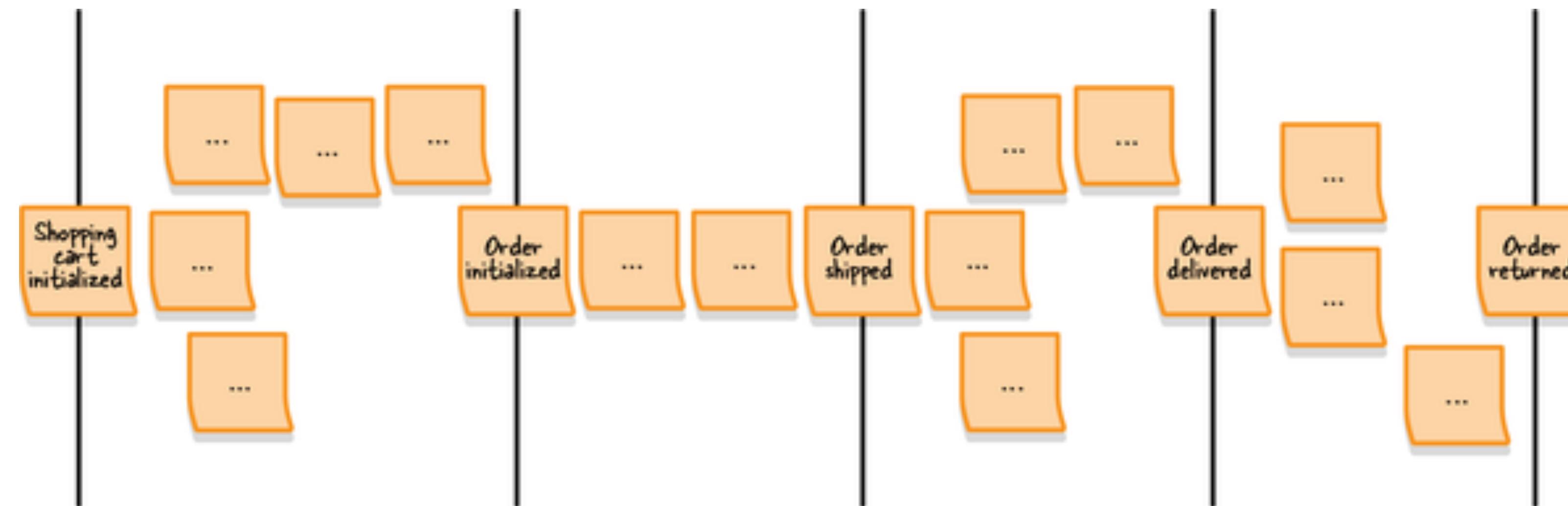
“Just keep in mind that the goal of the workshop is to learn as much as possible in the shortest time possible. We invite key people to the workshop, and we don’t want to waste their valuable time”

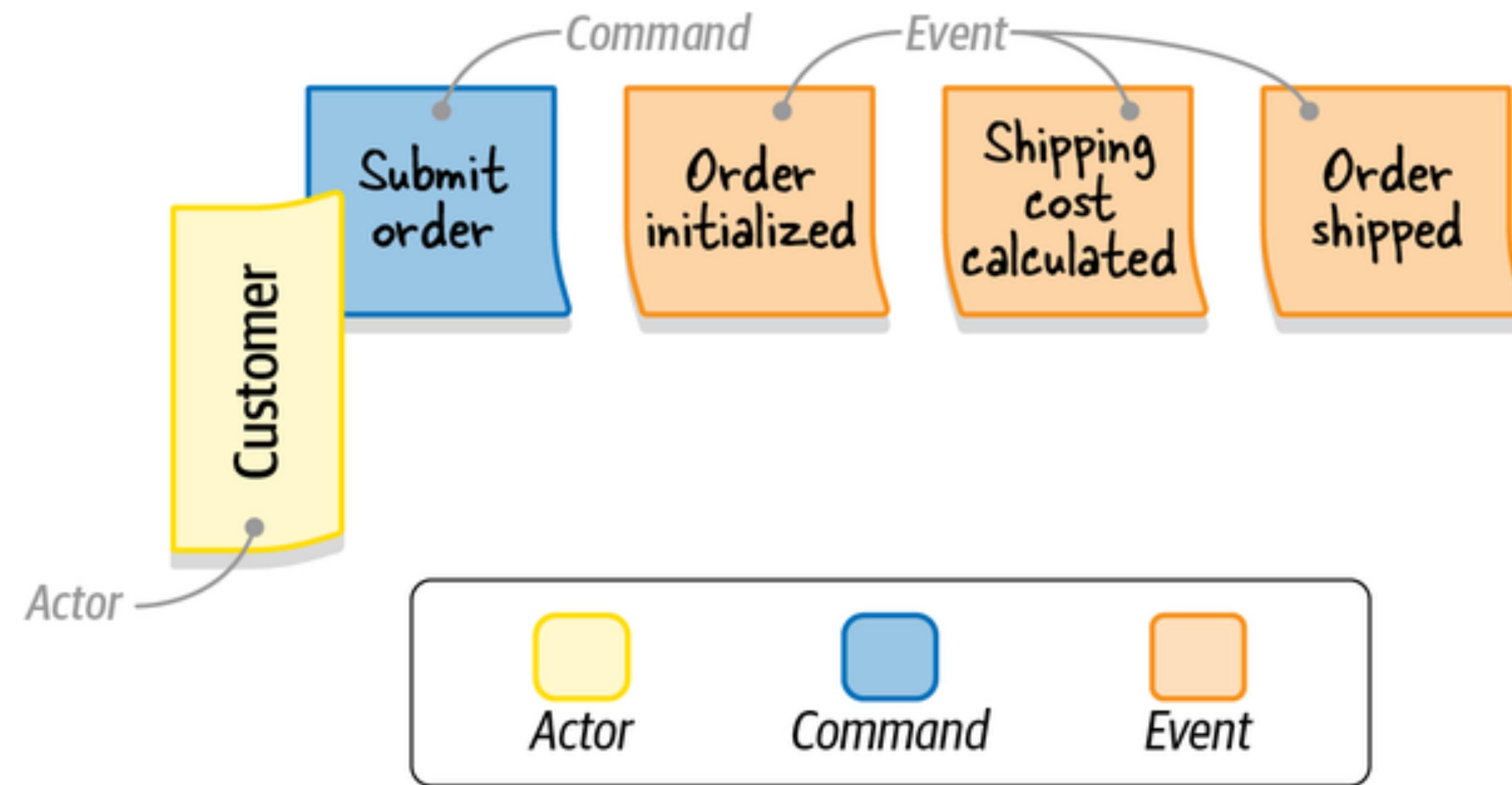
– Alberto Brandolini: Creator of the EventStorming workshop

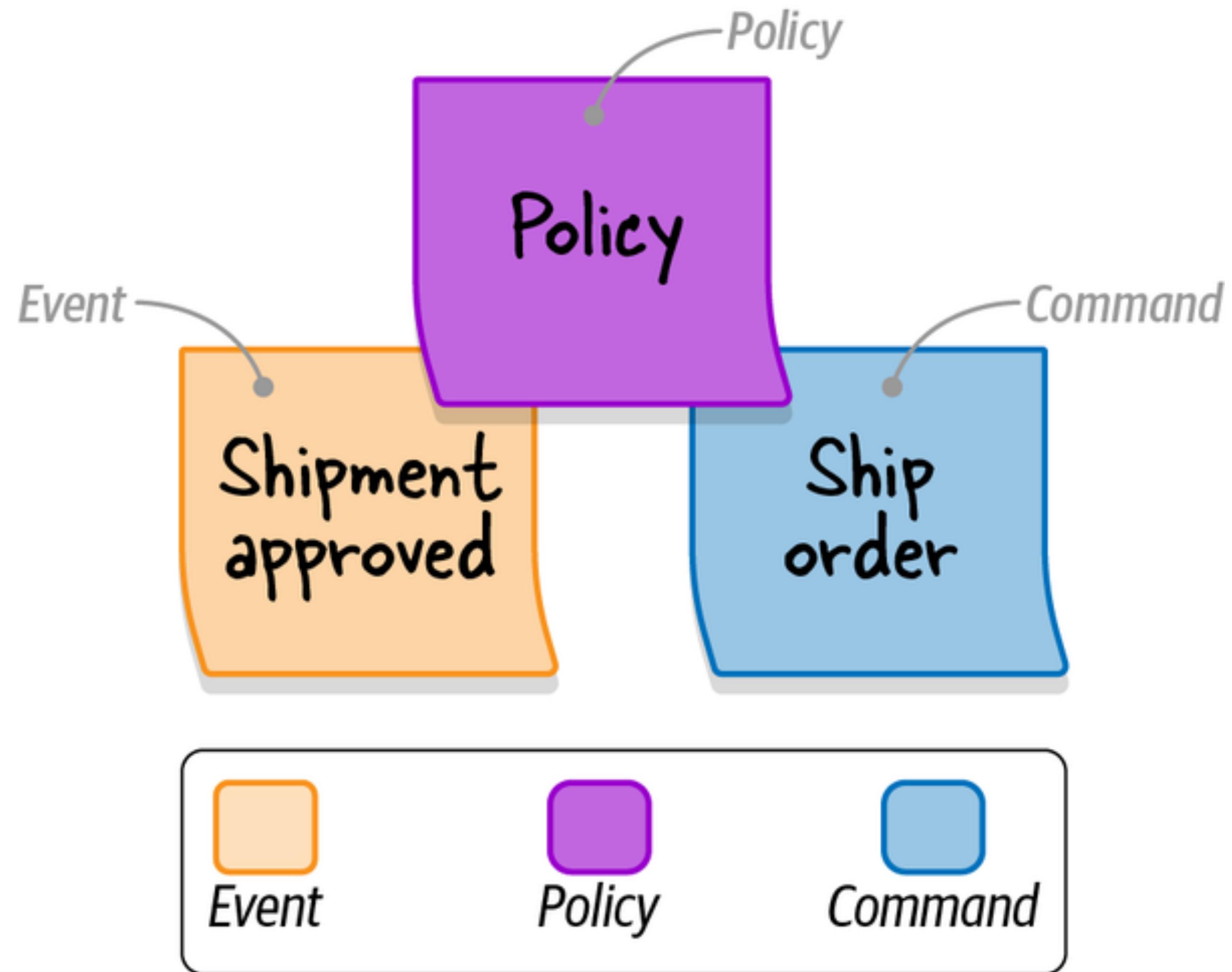


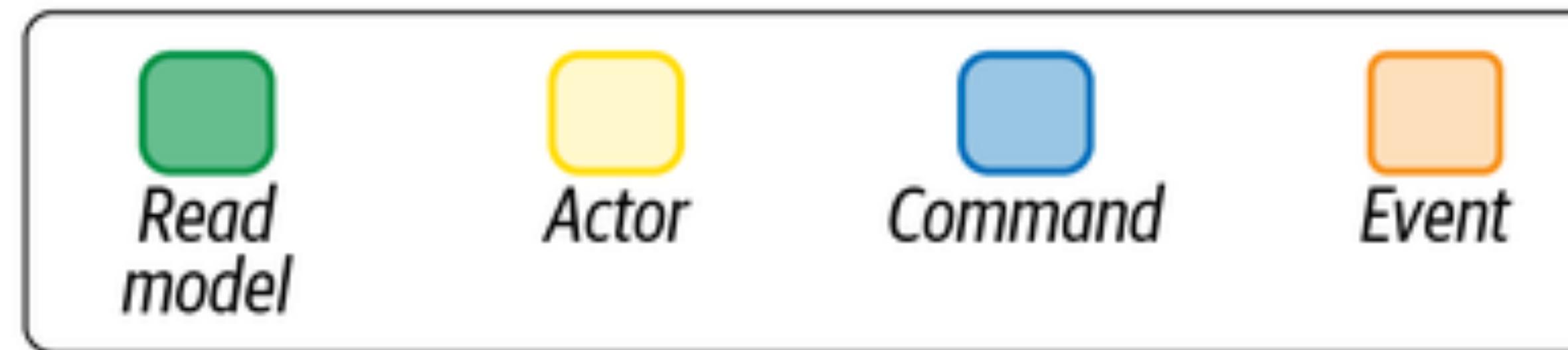
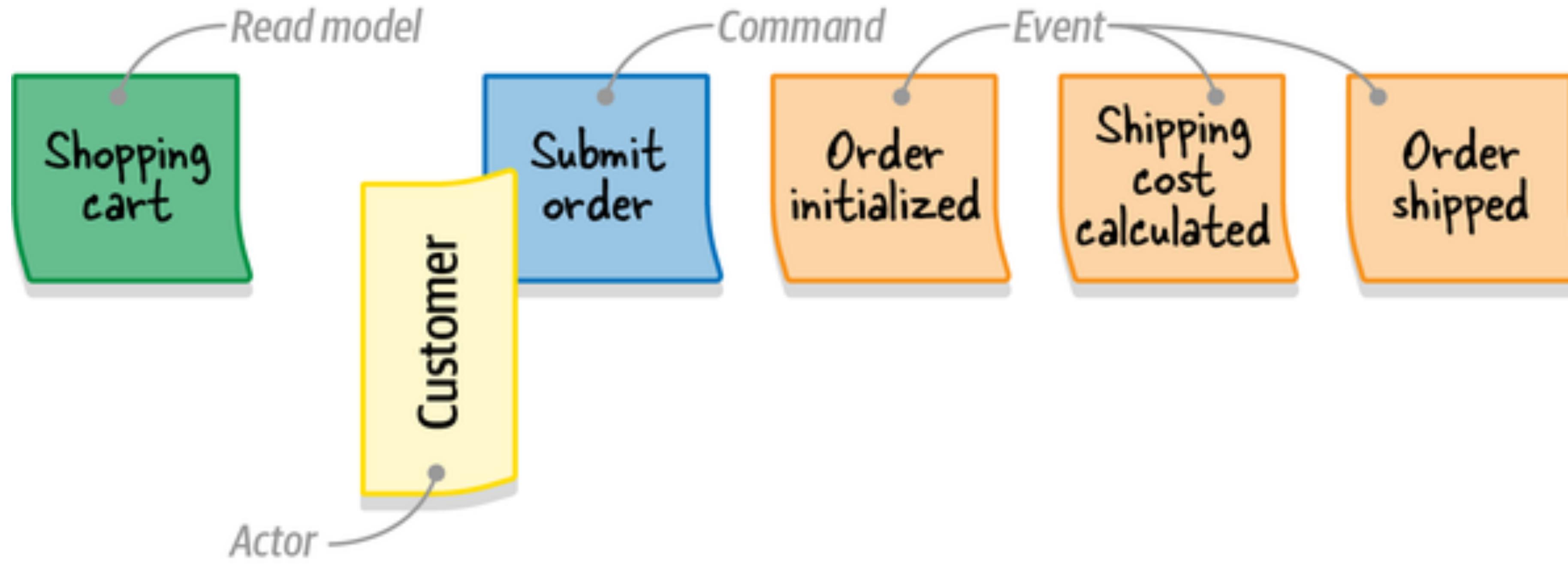








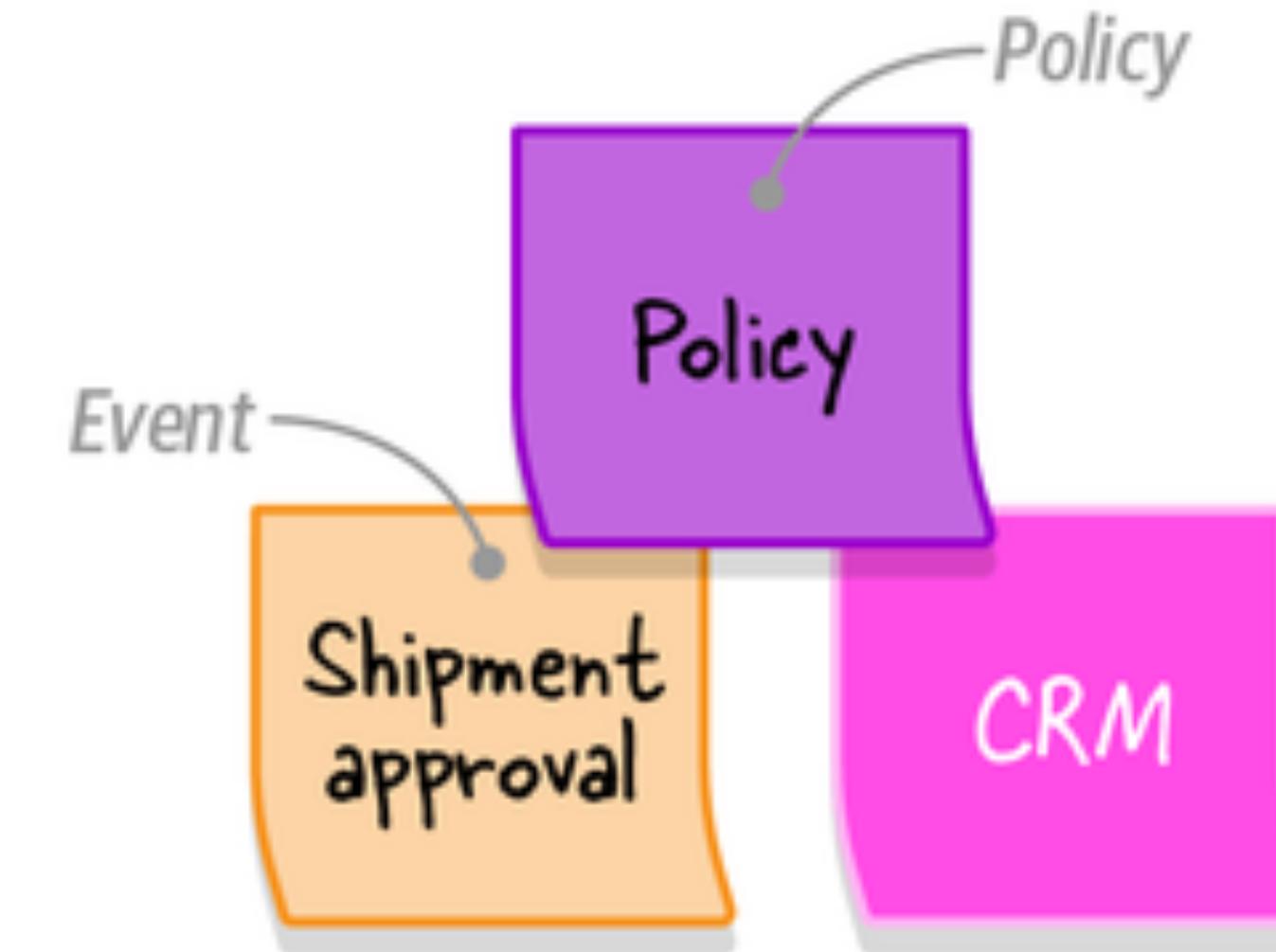


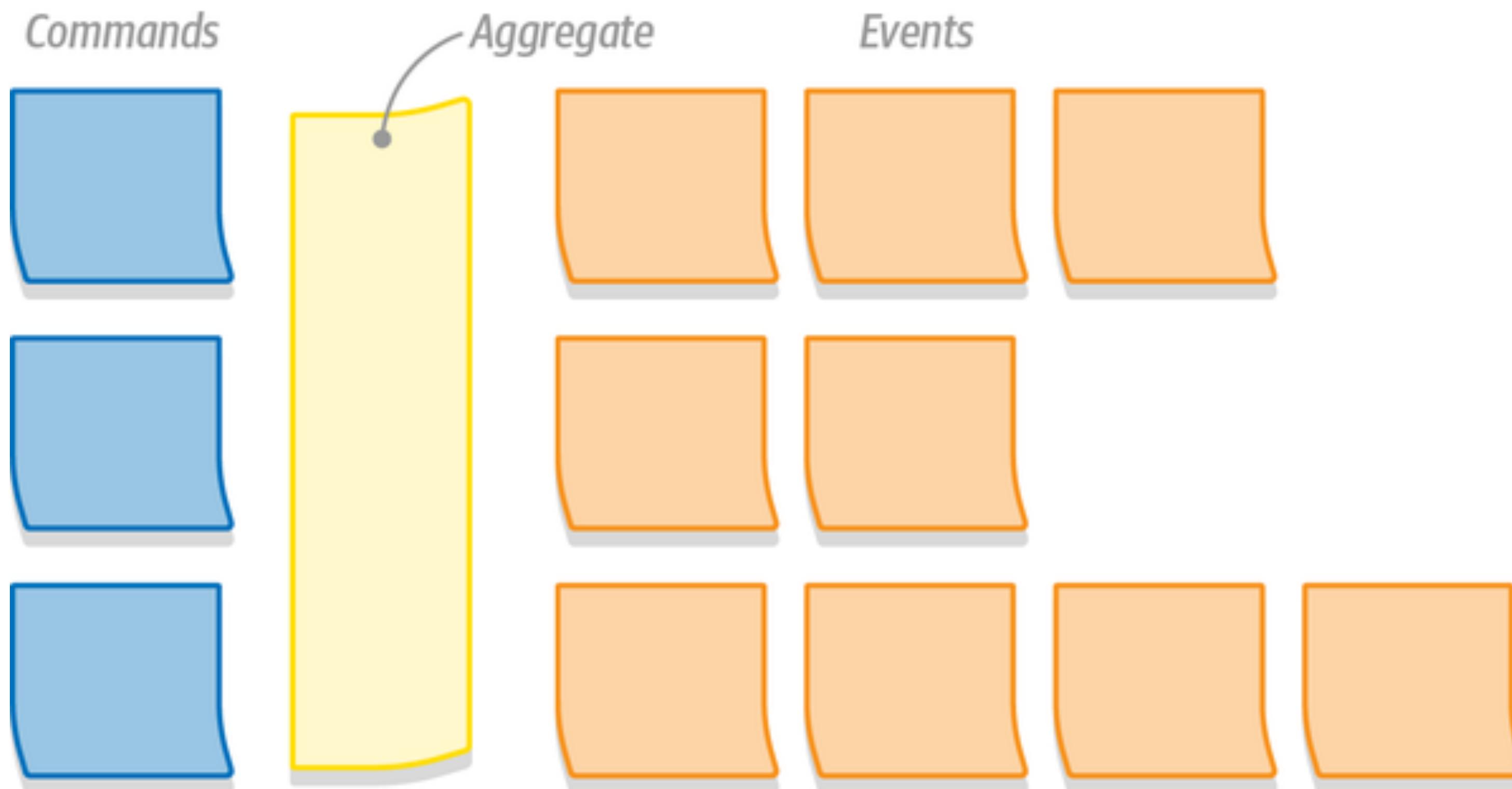


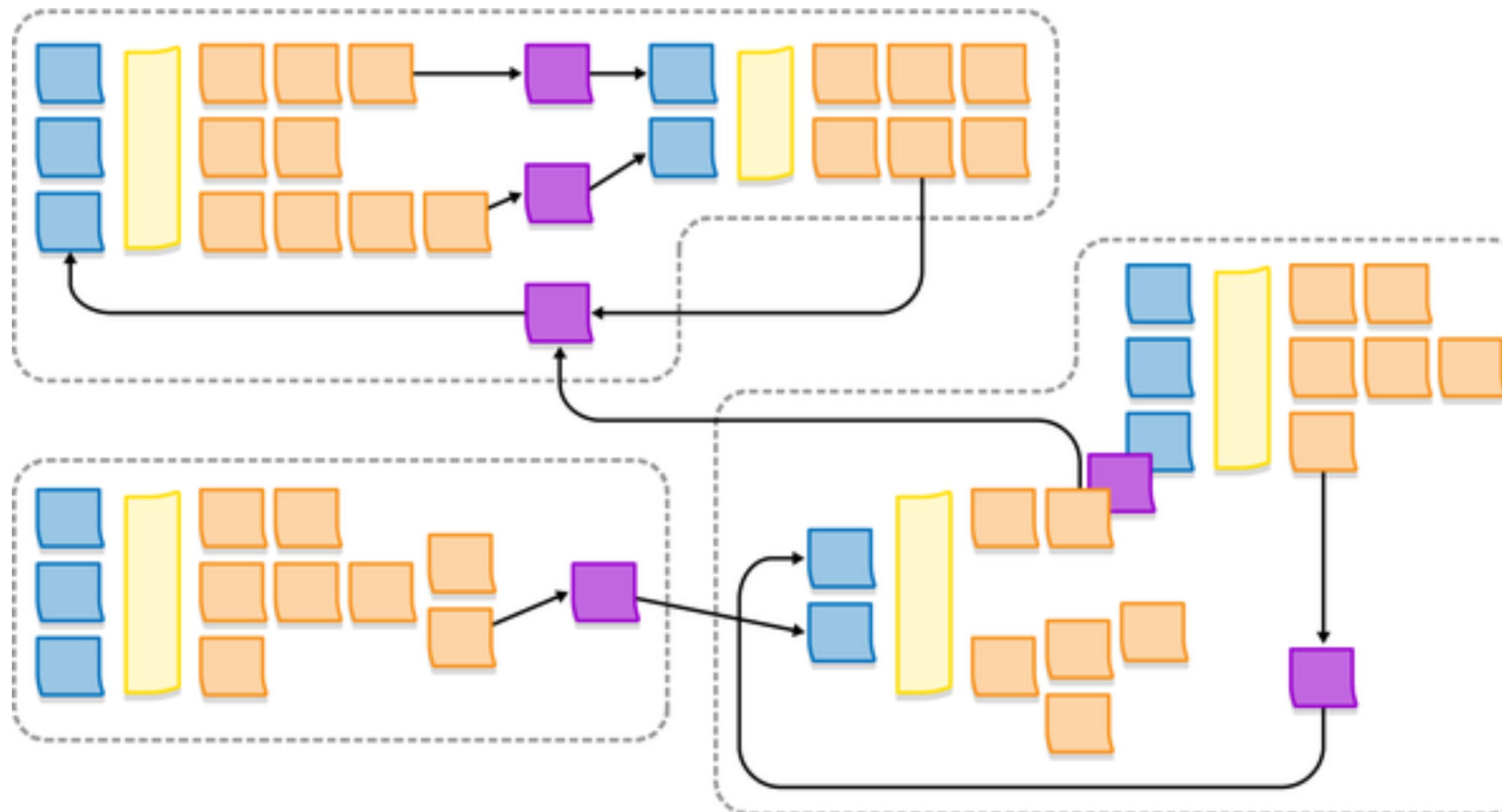
An external system executes a command



Notifying an external system







Why do you need to know DDD now?

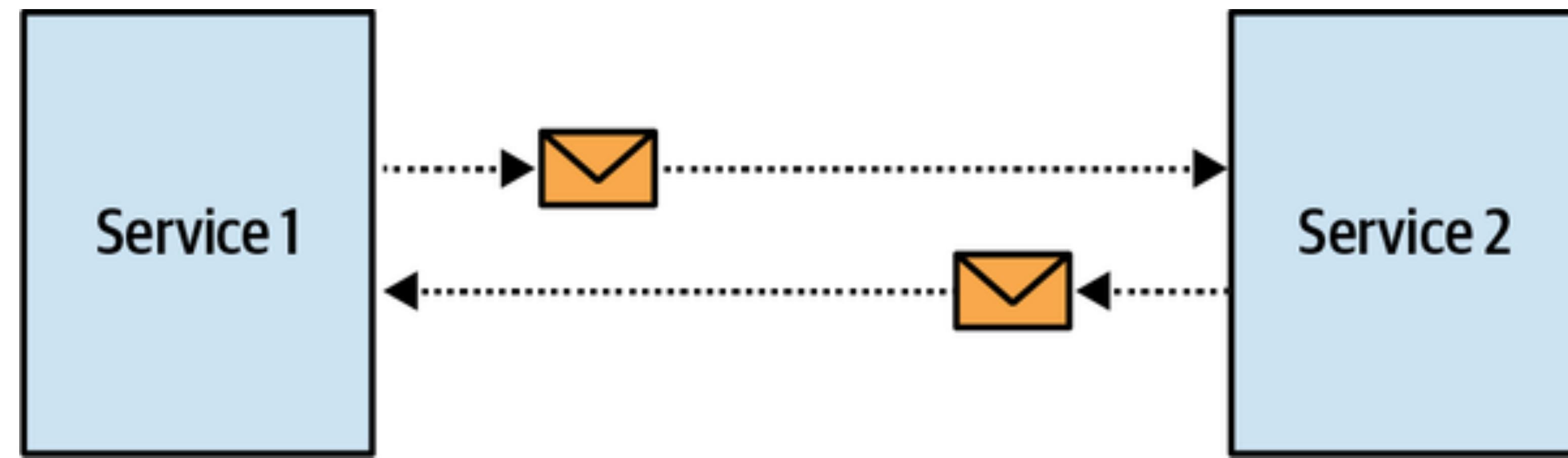


Why is DDD a must know today?

- Basis for Event Driven Architecture
- Basis for Data Management and Data Mesh
- Basis for Team Organization

Event Driven Architecture

- Architectural style in which a system's components communicate with one another asynchronously by exchanging event message
- Instead of calling the services' endpoints synchronously, the components publish events to notify other system elements of changes in the system's domain.
- The components can subscribe to events raised in the system and react accordingly



Two Types of Messages

- **Event**
 - A message describing a change that has already happened
- **Command**
 - A message describing an operation that has to be carried out

Event Sourcing

- Events are recorded in an append only store
- Events are described with discrete events like:
 - AddedItemToCart
 - RemovedItemFromCart
 - ClearedCart
- Therefore it can be used to materialize view, prepare the data for the purposes of being efficiently read by a UI or business analytics for consumption
- Remember the orange sticky notes?

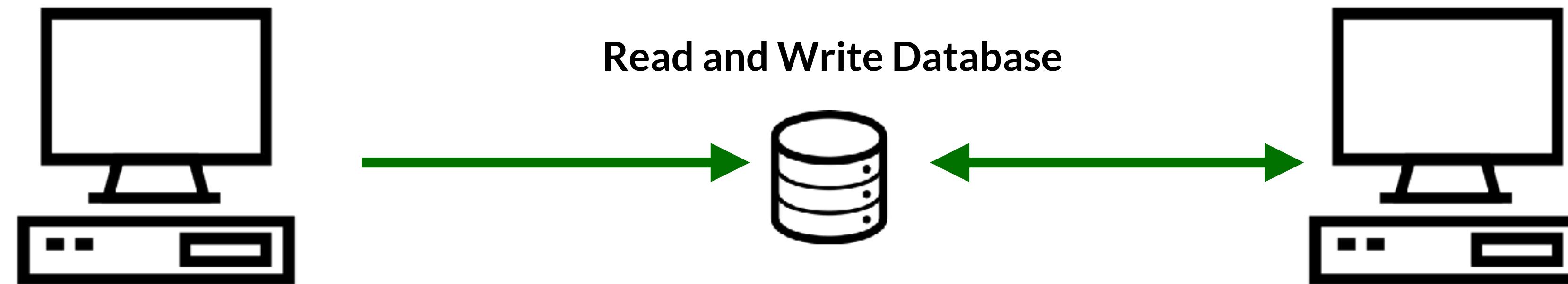


CQRS

- **Command and Query Responsibility Segregation**
- Separating reads, add-updates, for a databases
- Benefits include better performance, scalability, and security
- Better evolution over time

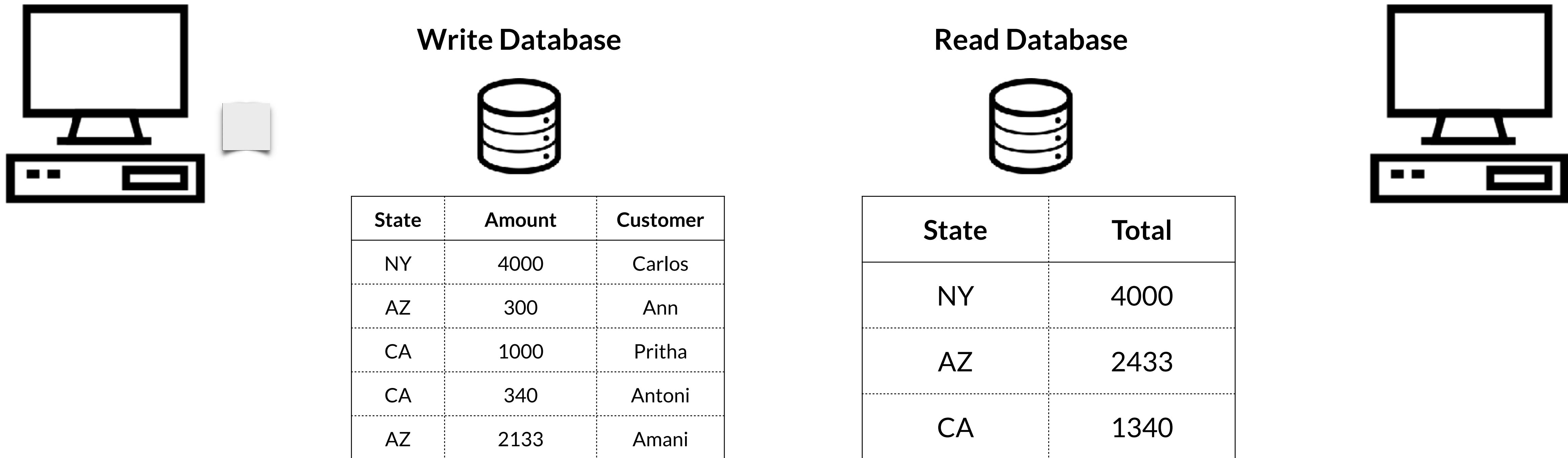
The Diagram

What if we need to query...
**SELECT state, count(*)
from orders
group by state, over and
over?**

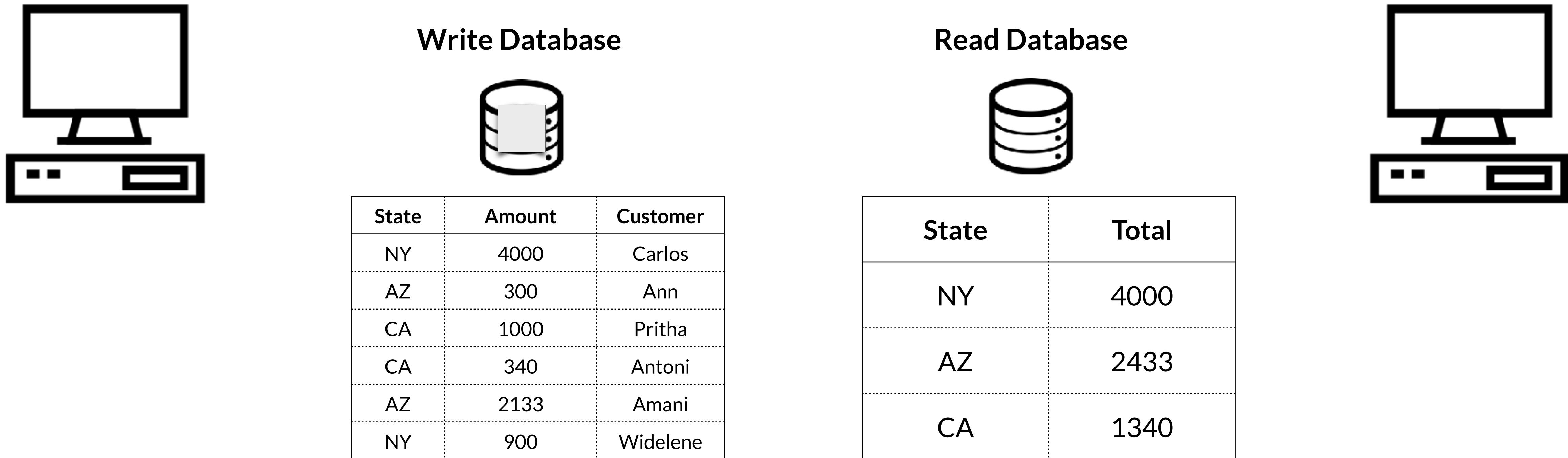


State	Amount	Customer
NY	4000	Carlos
AZ	300	Ann
CA	1000	Pritha
CA	340	Antoni
AZ	2133	Amani

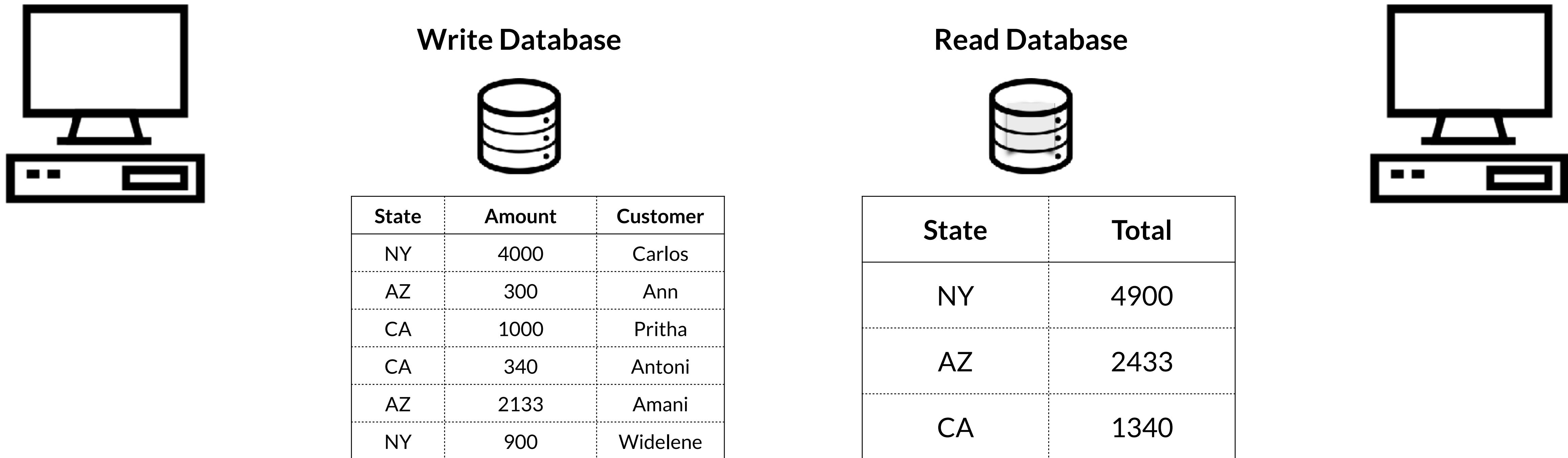
The Diagram



The Diagram



The Diagram



Data Mesh

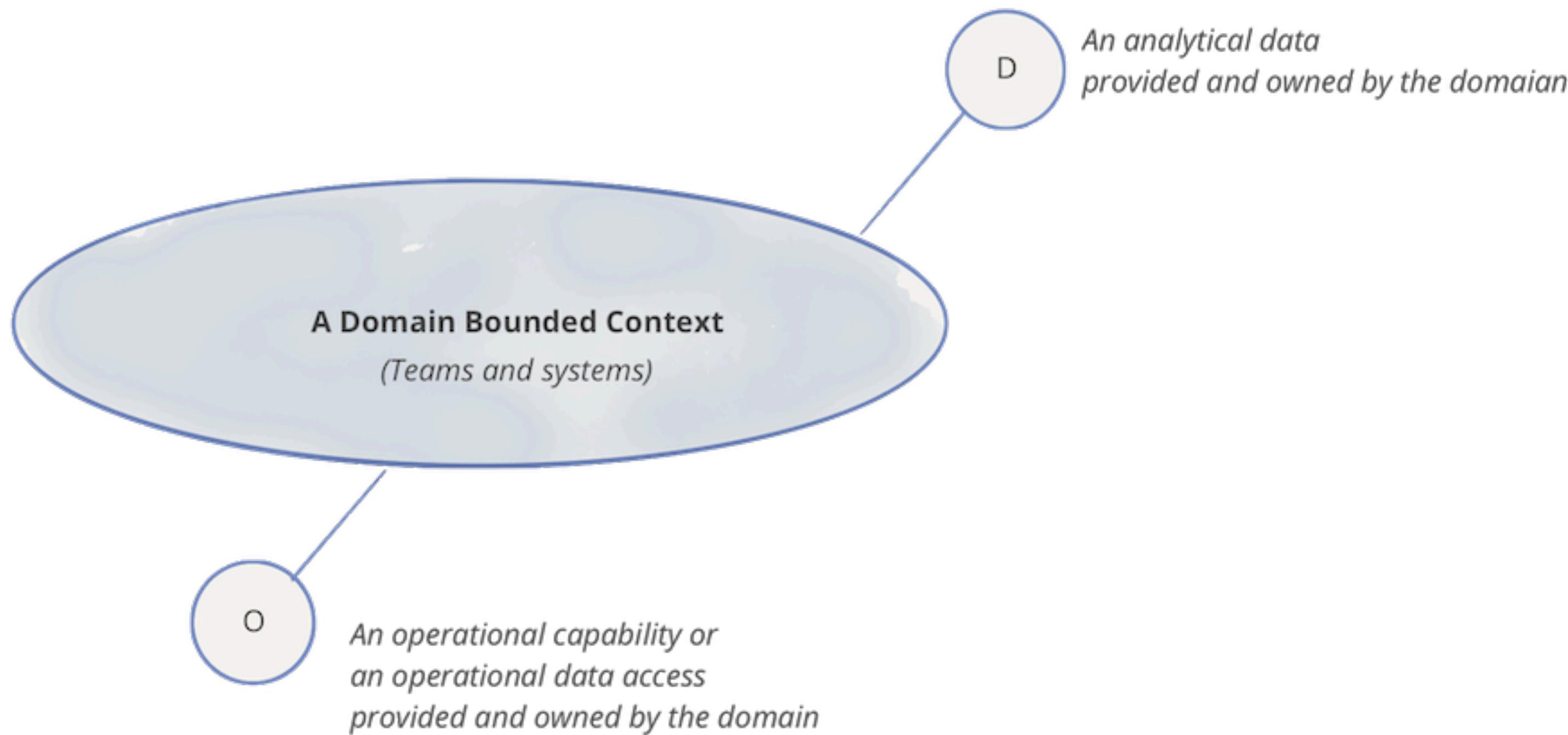
- Data can be siloed Database per Service. To circumvent, we created a data lake or shared database
- Problem is that the data lake, ended up as a data monolith
- By using a shared storage, we lost governance, and ownership
- We wanted to apply bounded context patterns, and did so with services, but lost the point with data.

Data Mesh

- There are two patterns: Database per Service, and Shared Database
- Data Mesh is an attempt at bringing not only the database but owning all data
- Data Mesh is about providing data ownership per context, and follows these principles:
 - Domain-oriented decentralized data ownership and architecture
 - Data as a product
 - Self-serve data infrastructure as a platform
 - Federated computational governance
 - The accountability of data quality shifts upstream as close to the source of the data as possible.

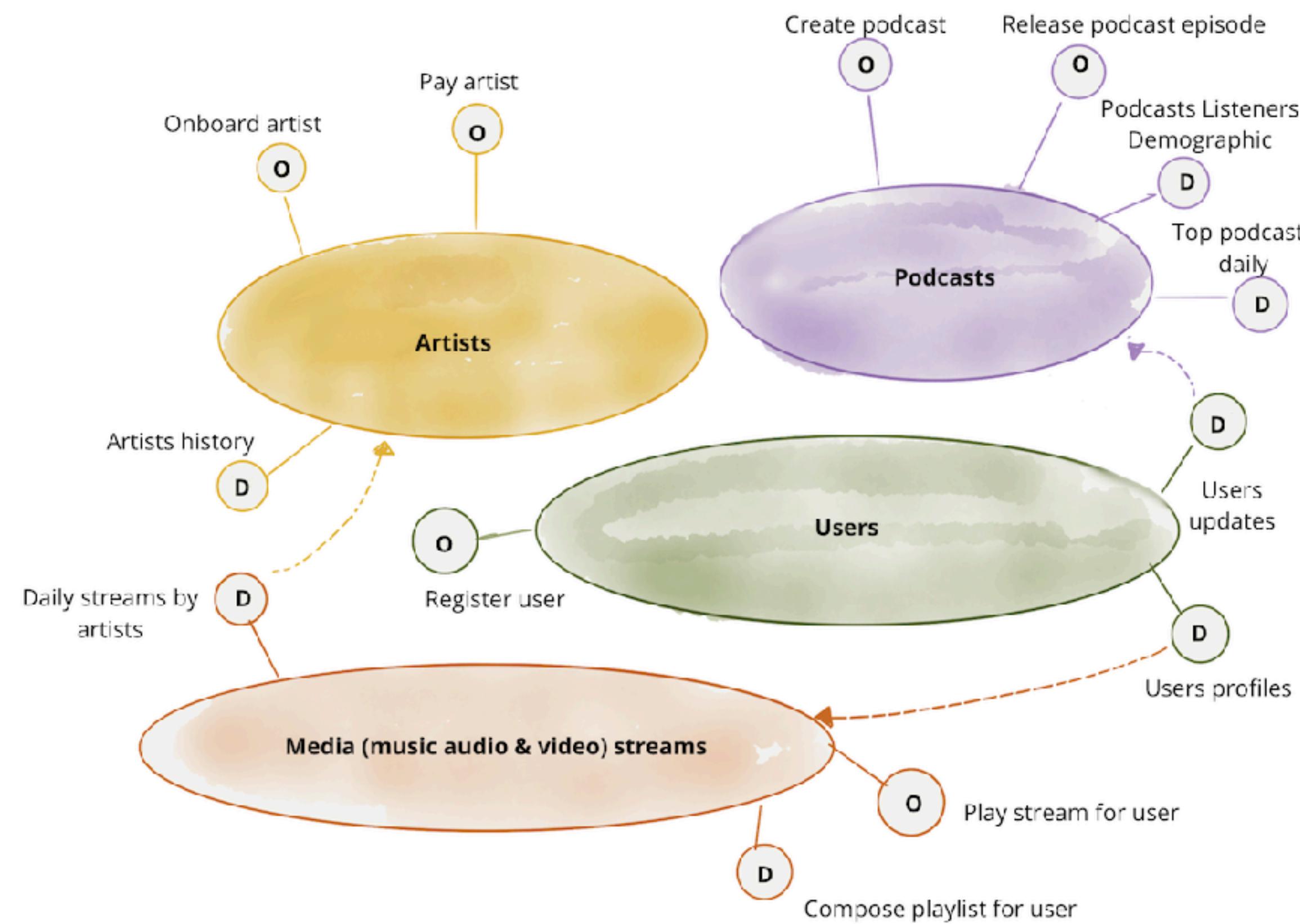
The corps was composed of all arms of the service, was **self-sustaining**, and could fight on its own until other corps could join in the battle. The corps itself was a headquarters to which units could be attached. **It might have attached two to four divisions of infantry with their organic artillery, it had its own cavalry division and corps artillery, plus support units.** With this organization a corps was expected to be able to hold its ground against, or fight off an enemy army for at least a day, when neighboring corps could come to its aid. "Well handled, it can fight or alternatively avoid action, and maneuver according to circumstances without any harm coming to it, because an opponent cannot force it to accept an engagement but if it chooses to do so it can fight alone for a long time."

The Diagram



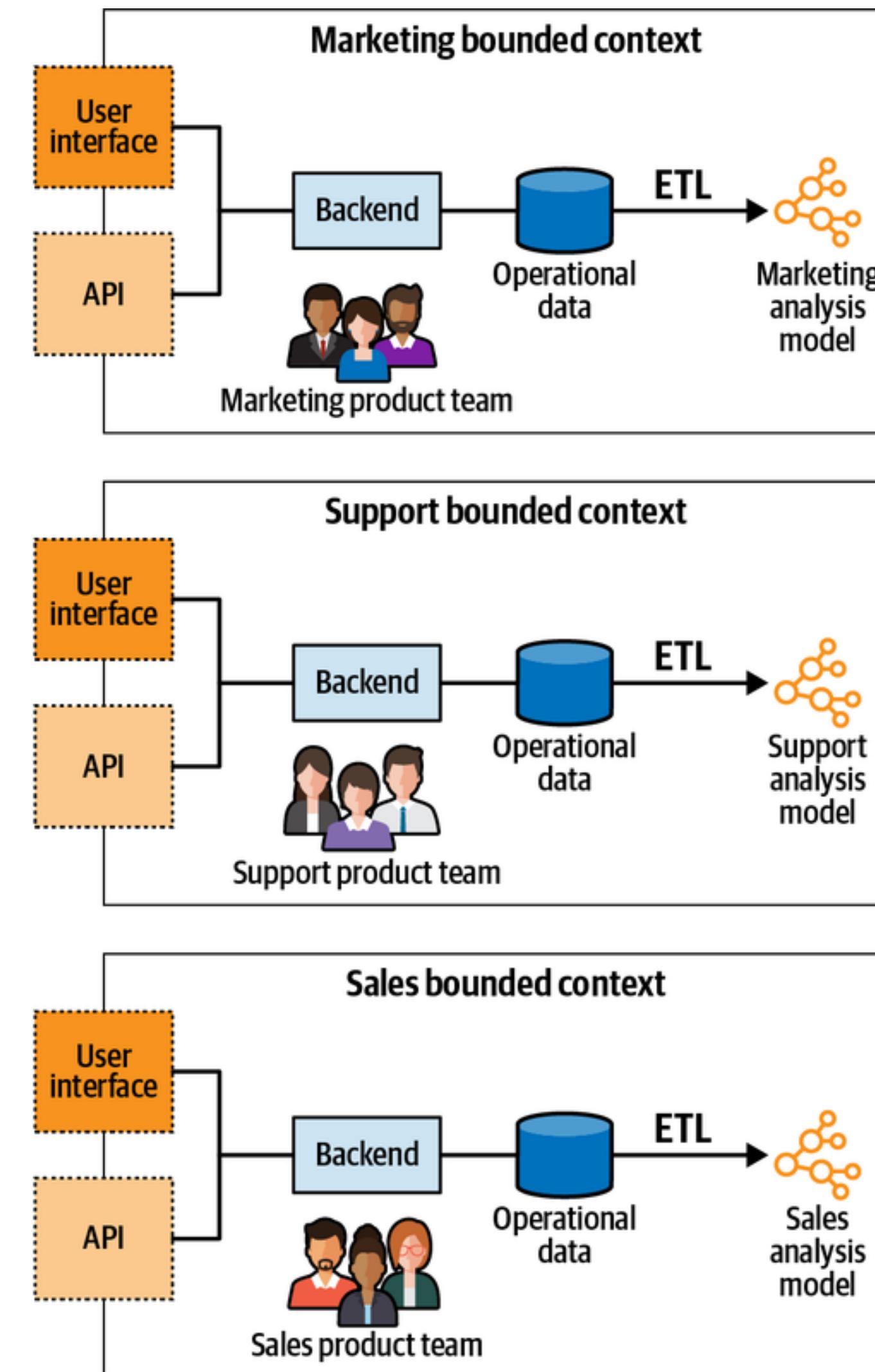
<https://martinfowler.com/articles/data-mesh-principles.html>

The Diagram

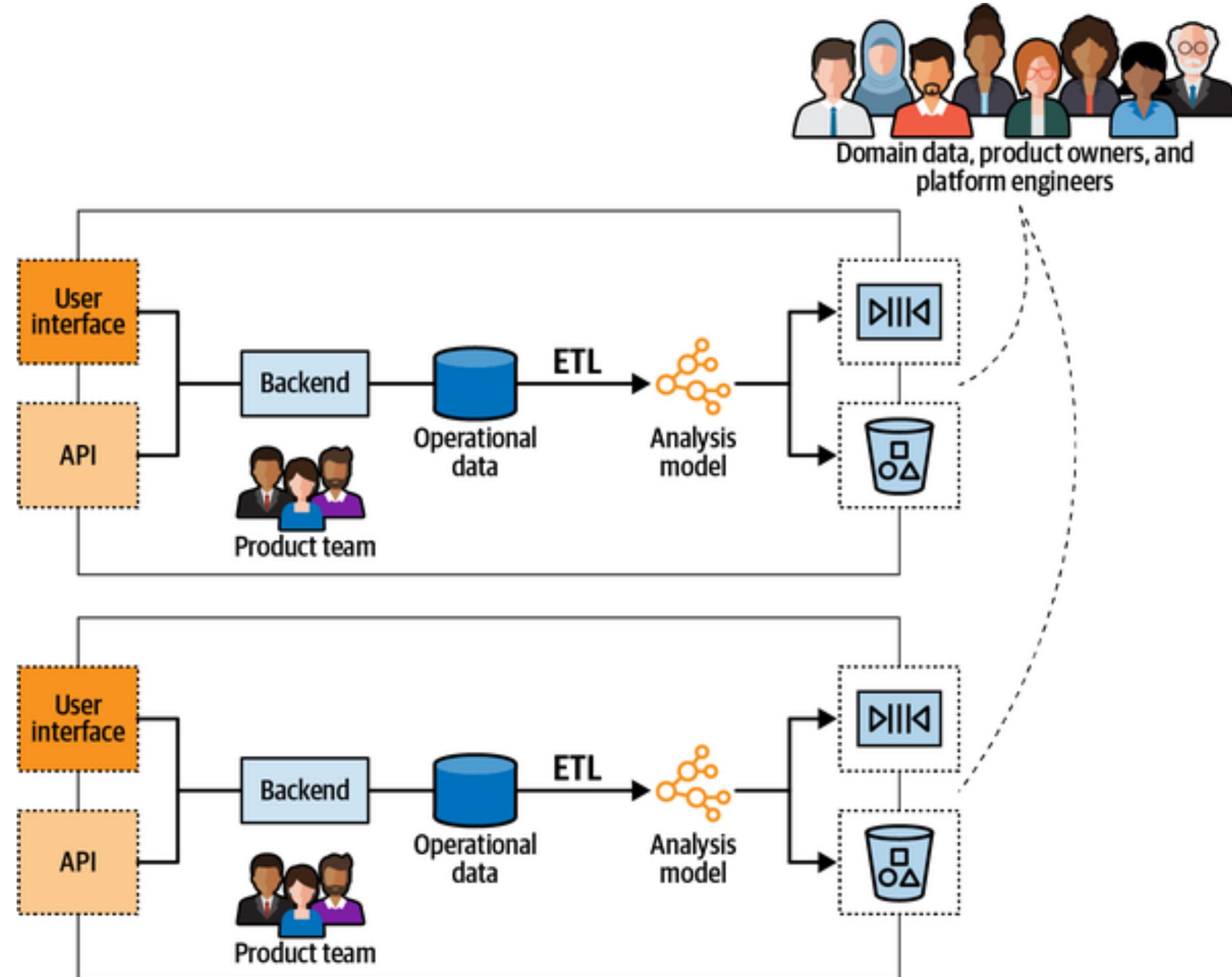


<https://martinfowler.com/articles/data-mesh-principles.html>

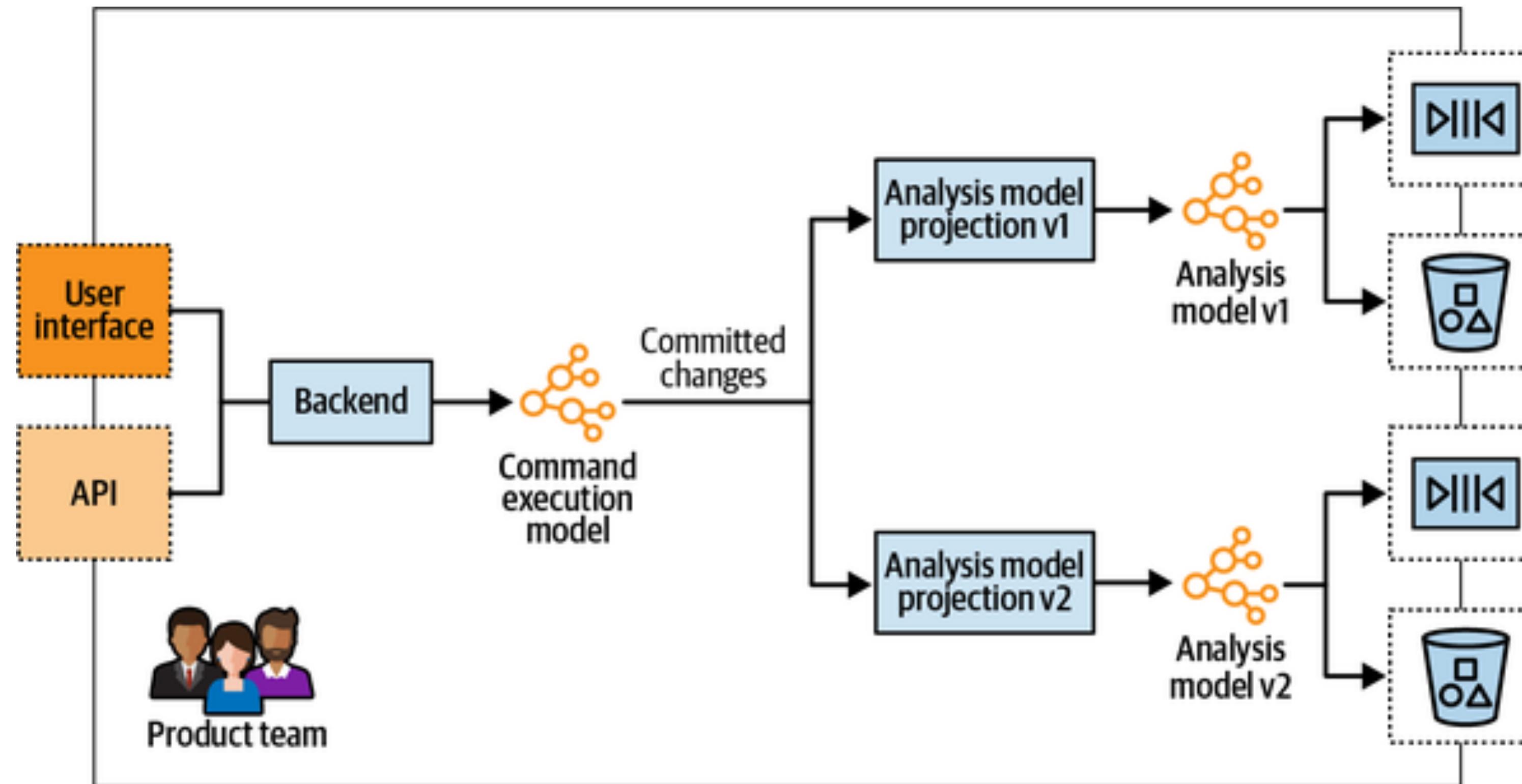
The Diagram



Federated Governance



Having different models of the data



O'REILLY®

Learning Domain-Driven Design

Aligning Software Architecture
and Business Strategy



Vlad Khononov
Foreword by Julie Lerman

TIME TO COMPLETE:

8h 32m

LEVEL:

Intermediate to advanced

SKILLS:

Domain-Driven Design

PUBLISHED BY:

O'Reilly Media, Inc.

PUBLICATION DATE:

October 2021

PRINT LENGTH:

342 pages

Thank You



- Email: dhinojosa@evolutionnext.com
- Github: <https://www.github.com/dhinojosa>
- Mastodon: <https://mastodon.social/@dhinojosa>
- Linked In: <http://www.linkedin.com/in/dhevolutionnext>